EFFICIENT PARALLEL ALGORITHMS FOR ERROR CORRECTION AND TRANSCRIPTOME ASSEMBLY OF BIOLOGICAL SEQUENCES

A Thesis
Presented to
The Academic Faculty

by

Vipin Sachdeva

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
May 2018

EFFICIENT PARALLEL ALGORITHMS FOR ERROR CORRECTION AND TRANSCRIPTOME ASSEMBLY OF BIOLOGICAL SEQUENCES

Approved by:

David A. Bader, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Kirk Jordan
IBM T. J. Watson Research Center
Cambridge, MA

Srinivas Aluru
School of Computational Science and Engineering
Georgia Institute of Technology

Umit V. Catalyurek
School of Computational Science and Engineering
Georgia Institute of Technology

Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: 13 March 2018

To my Family

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Next-generation sequencing technologies have led to a big data age in biology. Since the sequencing of the human genome, the primary bottleneck has steadily moved from collection to storage and analysis of the data. The primary contributions of this dissertation are design and implementation of novel parallel algorithms for two important problems in bioinformatics – error-correction and transcriptome assembly. For error-correction, we focused on k-mer spectrum based error-correction application called Reptile. We designed a novel distributed memory algorithm that divided the k-mer and tiles amongst the processing ranks. This allows any hardware with any memory size per node to be employed for error-correction using Reptile's algorithm, irrespective of the size of the dataset. Our implementational achieved highly scalable results for E.Coli, Drosophila as well as the human datasets which consisted of 1.55 billion reads. Besides an algorithm that distributes k-mers and tiles between ranks, we have also implemented numerous heuristics that are useful to adjust the algorithm based on the hardware traits. We also implemented an extension of our parallel algorithm further by using pre-generating tiles and using collective messages to reduce the number of point to point messages for error-correction. Further extensions of this work have focused to create a library for distributed k-mer processing which has applications to problems in metagenomics.

For transcriptome assembly, we have implemented a hybrid MPI-OpenMP approach for Chrysalis, which is part of the Trinity pipeline. Chrysalis clusters minimally overlapping contigs obtained from the prior module in Trinity called Inchworm. With this parallelization, we were able to reduce the runtime of the Chrysalis step of the Trinity workflow from over 50 hours to less than 5 hours for the sugarbeet dataset. We also employed this implementation to complete transcriptome of a 1.5 billion reads dataset pooled from different bread wheat cultivars. Furthermore, we have also implemented a MapReduce based approach to clustering k-mers which has application to the parallelization of the Inchworm

module of Trinity. This implementation is a significant step towards making *de novo* transcriptome assembly feasible for ever bigger transcriptome datasets.

CHAPTER I

INTRODUCTION

1.1 Big Data Computational Biology - Problems and Challenges

The term Big Data has become omnipresent in the last few years. The term first coined by Roger Magoulas from OReilly [40] media refers to datasets that pose inherent challenges both in terms of storing and analysis of this data. Data-intensive science combines the processing of tera- to peta-byte sized datasets with increased computational complexity. Increasingly, such problems can be observed in the life sciences domain where increasing amount of data from Roche 454, Illumina/Solexa and ABI SOLid platforms provide terabytes of DNA- and RNA-reads that need to be analyzed for insight into genes and transcriptomes. This insight is capable of altering the landscape of genetics. The sequence data generated by many of these machines consists of millions to billions of short DNA reads ranging from 50 to 150 base pairs in length. This has allowed the sequencing of whole genomes itself to be far cheaper than Sanger sequencing. However, this short read sequencing technology has come with its own issues: the short reads sequencing leads to enormous amounts of data for each run, which poses many challenges for its storage, analysis and interpretation. The bottleneck of these sequencing technologies thus lies not in the sequencing itself, but in the management and analysis of the data generated.

With the surge in sequencing data, genomics pipelines are omnipresent: both DNA-seq and RNA-seq pipelines are widely used for several purposes in genomics. A DNA-seq pipeline is composed of several steps leading to DNA assembly from short reads data from sequencing machines: in case of availability of a reference genome, align- ment is the first step that maps DNA-seq reads to a reference genome. Many of the sequence alignment tools are either hash-based or work with the Burrows-Wheeler transform. In hash- based methods such as MAQ [5], as a first step, a hash table is constructed on the set of input reads, or on the reference genome. The hash table is then scanned using the

reference genome or vice versa. Burrows-Wheeler tranform methods such as in Bowtie [32] perform alignment using suffix arrays constructed from Burrows-Wheeler transformed sequences. Assembly algorithms starts from DNA-seq reads to reconstruct the original DNA sequence computationally which generates large, continuous regions of DNA sequence. Many alignment tools already provide the functionality to perform the assembly after the read alignment such as MAQ. This step only works for organisms with a reference genome. De Novo assembly is carried out with either the following techniques: overlap graph [48] or the de Bruijn graph [53]. The overlap graph calculates all the pair-wise overlaps between the reads and put this information in a graph. This graph is then changed to a consensus sequence of contigs using other proprietary softwares such as Arachne. This approach can be very computationally intensive. De Bruijn graphs are used by most assemblers, and work primarily with k-mers which are smaller overlapping sub-sequences of reads. The de novo assembly can also be used to resolve more complex regions with a reference genome. Following these steps, basic quality control is performed which involves formatting the aligned reads in a conventional format (Sequence Alignment/Map (SAM) or the binary version of it, BAM) [35]. This leads to a sorted and indexed file in BAM format that is subsequently used to more advanced quality control procedures. Following the quality control, SNPs, insertions-deletions are detected . Many algorithms are used for detection of SNPs from next-generation sequencing data; the variants are reported in variant call format (VCF), with information on each variant. Subsequent to the calling of the variants, they are annotated as well. This is an example of a pipeline in analysis of DNA-sequence data.

RNA-seq pipeline is even more complex; transcriptomics gives information on which genes are expressed in a given cell type, under given conditions, at a given time point. The number of reads which map to a given gene or isoform is a direct measure of the expression level. Thus, transcriptomics is a major route to the study of gene expression, and is rapidly re- placing microarrays as the method of choice. As in genome sequencing, the cDNA libraries are chopped up into millions of short reads which are then sequenced. The computational task is then to re-assemble these sequenced reads into a set of transcripts corresponding to gene products. As the sequencing technology improves, the computational

step is becom- ing the principal bottleneck. A dataset consists of a large set of sequenced reads (provided as a FASTA or FASTQ file) which can have a size on the same order as for genome se- quencing. Unlike genome sequencing though, an organism can have multiple transcriptomes corresponding to different cell types or conditions. The recommended practice is to sequence these together into a consensus transcriptome [21], and thus the size of the dataset is multiplied by the number of experiments considered. Transcriptome assemblies are followed by further downstream analysis including comparing transcriptomes across samples, tran- script abundance estimation, analysis of differentially expressed transcripts and prediction and functional annotation of transcripts. Transcriptomics gives information on which genes are expressed in a given cell type, under given conditions, at a given time point. The number of reads which map to a given gene or isoform is a direct measure of the expression level. Thus, transcriptomics is a major route to the study of gene expression, and is rapidly replacing microarrays as the method of choice. Transcriptome assembly works on very large datasets and can require considerable compute resources. It is a high-performance computing (HPC) problem.

Genomics has thus become a big data science problem. Zachary et al reference here have shown that genomics data is projected to be 2-40 EB per year in the 2025 timeframe. As shown in DNA- and RNA-seq pipelines, the analysis and data of genomics is very heterogenous. Steps such as variant calling and all-to-all comparisons which are extremely popular in genomics pipelines are projected to consume trillions of CPU hours. Although the data from sequencing experiments has become cheaper and has grown by orders of magnitude, many of the commonly-used bioinformatics tools were developed for desktop applications, and most assume a shared memory architecture. This especially creates a problem for memory-intensive problems in genomics such as RNA-seq and error correction. Software created for these problems originally struggles with todays large datasets, taking days to run routine jobs, and often exceeding the available memory. At the same time, the trend for HPC is towards more parallelism, with larger numbers of lower power nodes [29]. Adapting bioinformatics tools for multi-node distributed memory architectures is thus essential. With a few notable exceptions [72][27][3], distributed memory architectures are

not well supported by existing bioinformatics tools.

1.2 Overview of Dissertation

1.2.1 Distributed Memory Algorithms for Transcriptome Assembly and Error-Correction

As part of this dissertation, we have focused on designing and implementing distributed memory algorithms for two important problems in genomics - transcriptomic assembly and error-correction. For transcriptomic assembly, we have chosen to look at the Trinity pipeline [19] which is one of the most popular packages for transcriptomic assembly [20]. It was originally created for de novo assembly, although there is now a protocol for assembly against a reference genome. De novo assembly constructs the transcriptome solely from the available reads, and is useful when there is no reference genome available, or if there are likely to be large structural variations (e.g. in cancer cell lines) [21]. Trinity is a pipeline implemented in Perl whose modules assume a shared-memory architecture.

Error-correction is another data-intensive computational biology problem; next-generation sequencing (NGS) technologies have steadily replaced Sanger sequencing as the preferred method of genome sequencing [69]. They provide far more information than the Sanger method, at a much lower cost per DNA base. However, NGS methods suffer from two critical disadvantages: they produce significantly shorter pieces of genome called reads that have to be assembled together, and are more error-prone than the Sanger method. Error-correction is thus a crucial step of many bioinformatics pipelines. It has been shown that transcriptome assembly results are improved by error-correction [39]. There are several error-correction methods but K-mer spectrum based error correction methods have proven to be a superior error-correction method. K-mer spectrum methods however depend on the k-mer spectra of the reads. The first step of the k-mer spectrum methods consist of generating k-mer spectra from the reads file. The k-mer spectra for high k-lengths and reads file consisting of billions of reads is often in the hundrededs of gigabytes range and often cannot fit into the memory of a single node. Thus, efficient methods to distribute k-mer spectra over multiple nodes and use message passing to exchange k-mer information are important. We have designed and implemented efficient k-mer processing method as part of parallel

implementation of Reptile. This distributed k-mer spectra is utilized in the error-correction phase, where multiple ranks exchange messages for k-mers and tiles on other ranks.

Efficient distributed k-mer processing also finds it's utility in other problems such as metagenomics. Our distributed k-mer processing implementation is part of a library; we are using this functionality to ascertain the similarity between unknown metagenomic samples.

1.2.2 Organization

The rest of the dissertation is organized as follows: in Chapter 2, we discuss the distributed memory implementation of Reptile [82], a scalable and accurate spectrum based error-correction method. Reptile uses both k-mer and adjoining k- mers (called tiles) information along with the quality scores of bases to correct substitution-based errors from next generation sequencing machines. Previous approaches [70] [28] to parallelize Reptile have replicated the spectrums on each node which can be prohibitive in terms of memory needed for huge datasets. Our approach distributes both the k-mer and the tile spectrum amongst the processing ranks, relying on message passing for error correction. This allows hardware with any memory size per node to be employed for error-correction using Reptiles algorithm, irrespective of the size of the dataset. As part of our implementation, we have also implemented several heuristics which can be used to run the algorithm optimally based on the advantages of the hardware used. We present our results on IBMs BlueGene/Q architecture for the E.Coli, Drosophila and the human datasets showing excellent scalability with increasing number of nodes. Using 256 nodes of BlueGene/Q, we are able to error correct E.Coli and Drosphila datasets in less than 200 seconds and 600 seconds respectively. The human dataset consisting of 1.55 billion reads is corrected in a little more than two hours using 1024 nodes of BlueGene/Q. All three datasets are corrected with Reptiles memory intensive algorithm with less than 512 MB per process. We also discuss an alternative strategy for Reptile parallelization which pre-generates tiles per batch of reads, and subsequently uses collective communication to lookup the counts of these tiles in advance. Towards the end of the chapter, we also detail some of the current work with exploring

k-mer spectrum for ascertaining metagenomics similarity. As part of the solution for error-correction, my implementation included steps for computing distributed k-mer spectra. We have repackaged this computation of distributed k-mer spectra into a separate library and are employing this in problems in metagenomics. In metagenomics, the distributed k-mer spectra is being used to compute Bray-Curtis coefficients between metagenomic samples to ascertain their similarity (or dissimilarity). In Chapter 3, we summarize the design and implementation of the distributed memory algorithms for the Trinity pipeline, in particular the GraphFromFasta and ReadsToTranscripts modules. Trinity is a pipeline implemented in Perl which wraps a number of underlying programs implementing different stages of the assembly [26]. In its evaluation, Trinity has been found to be accurate in transcriptome assembly under a variety of conditions, but with high runtimes [85]. The pipeline is very heterogenous in its computational requirements, with early stages requiring large amounts of memory, and later stages being more CPU-intensive. Trinitys assembly pipeline consists of four consecutive modules: Jellyfish, Inchworm, Chrysalis and Butterfly. The modules are separate executables, written in different languages. The Chrysalis step itself is composed of separate submodules including Bowtie [32], GraphFromFasta and ReadsToTranscripts. Previous attempts to speed up Trinity have focused on using OpenMP threads in a shared-memory architecture and reducing I/O operations [23]. We present results here for the Chrysalis module, aimed at speeding up this section of the pipeline by spreading the load across multiple distributed nodes, working seamlessly with the already existing OpenMP implementation. Parts of GraphFromFasta and ReadsToTranscripts which were written in OpenMP for shared memory have been changed to a hybrid implementation using MPI across distributed nodes, and OpenMP threads within a node. Furthermore, we have also developed a MapReduce based implementation of k-mer clustering algorithm which has application to the parallelization of the Inchworm module of Trinity.

1.3 Contributions

The central theme of this dissertation is the design and implementation of distributed memory algorithms for two important problems in genomics - transcriptome assembly and

error-correction. The key contributions of our work are as follows:

- *Novel parallel algorithms for transcriptome assembly applied to the Trinity pipeline.* Trinity is a highly popular RNA-seq pipeline composed of several heterogenous modules. As part of this work, Trinity's Chrysalis modules were implemented for a distributed-memory architecture. This implementation is now part of the Trinity pipeline. We have subsequently used the Trinity pipeline to perform de novo assembly of a 1.55 billion reads dataset from wheat cultivars. Furthermore, we have also implemented a MapReduce based approach to another Trinity module Inchworm.

- *Distributed-memory parallelization of a spectrum based error-correction application Reptile.* Previous implementations of k-mer spectrum based error-correction methods have replicated the k-mer spectrums on each process or node. This leads to very high memory requirements per node. Our approach uses a distributed memory time and memory scalable parallelization of the Reptile error-correction code.

  - *Numerous heuristics to improve performance of Reptile to be used based on the machine and dataset characteristics.* As part of our distributed memory implementation of Reptile error-correction code, we have also implemented several heuristics which can be used based on the traits of the dataset and the hardware. For memory-constrained architectures, we depend on a completely partitioned k-mer spectrum, while for architectures with high memory requirements, we replicate the k-mer and the tile spectrum partially on each node for better results.

  - *An alternative parallel implementation that relies on pre-generation of tiles and collective communication.* We have also added an additional algorithm for Reptile error-correction code: this algorithm pre-generates the tiles needed for error-correction and uses collective communication to reduce point to point messaging requirements during error-correction phase.

  - *A library framework to compute distributed k-mer spectrum and it's applications to metagenomics.* Leveraging the work from the parallel Reptile implementation,

we have implemented a library framework for computing distributed k-mer spectrum. This framework has been extended to evaluate distributed Bray-Curtis measure which can be used to ascertain dissimilarity of metagenomics samples.

CHAPTER II

EFFICIENT PARALLELIZATION AND COMPARISON OF ERROR CORRECTION CODES

Next-generation sequencing (NGS) technologies have steadily replaced Sanger sequencing as the preferred method of genome sequencing [69]. They provide far more information than the Sanger method, at a much lower cost per DNA base. However, NGS methods suffer from two critical disadvantages: they produce significantly shorter pieces of genome called reads that have to be assembled together, and are more error-prone than the Sanger method.

Error correction for datasets from NGS technologies is one of the most important steps for correct assembly results. The errors associated with NGS technologies can be classified into substitution errors which happen when a single base is altered to a different base, and insertion-deletion errors in which the entire region of a genome consisting of several characters has been changed. Correction of these errors greatly enhances the performance of many subsequent steps using this data such as de novo genome and RNA sequencing, re-sequencing and metagenomics besides others [39]. Error correction methods are broadly distinguished into k-spectrum, suffix tree based and multiple sequence alignment methods [81]. K-spectrum methods decompose reads into the set of all k-mers. In this approach, erroneous k-mers are converted to the consensus k-mer (or the highest frequency k-mer). Suffix tree based methods generalize the k-mer based approach and multiple sequence alignment methods identify reads co located on the unknown reference genome by using k-mers as seeds. Please refer to [81] for a comprehensive evaluation of error correction of reads.

Reptile [82] is a scalable substitution error correction method that has been shown to outperform several other error correction methods [81]. Reptile is broadly based on the k-mer spectrum approach, but also uses adjoining k-mer information (called tiles) to reliably suggest error corrections. Since Reptile relies on both k-mer and tile spectrum for more

accurate error correction, there are memory limits on the size of datasets it is able to error-correct. The key contributions in this chapter are as follows:

- A novel distributed memory algorithm for parallelization of Reptile which distributes both the k-mer and tile spectrum amongst the processing ranks. Our approach allows hardware (with any memory size per node) to be employed for error correction of any dataset using Reptile's algorithm. During error correction of a read, it is expected that a rank will need the k-mers and tiles it does not store in its own local memory. We rely on message passing for such k-mers and tiles.

- Besides a distributed k-mer and tile spectrum, our implementation also has support for numerous heuristics that allow parallel Reptile to use the features of the underlying hardware efficiently.

- We use our algorithm and implementation with improvements and heuristics to present highly scalable results for error-correction of E.Coli, Drosophila and human datasets with less than 512 MB per process on IBM's BlueGene/Q architecture [22].

- We have implemented another algorithm for parallelization of Reptile that generates the tiles required by the processes beforehand, and subsequently uses collective communication for tiles of non-corrected reads to get the counts of the tiles needed for error correction. During error correction phase, the counts for tiles of reads which need to be error corrected are exchanged through point to point messaging as in the previous algorithm.

2.1 Reptile and other Error-Correction codes

2.1.1 Reptile algorithm description

Reptile is a spectrum based substitution error-correction method; instead of only relying on the k-mer spectrum (consisting of k-mers), Reptile also constructs and subsequently uses another spectrum consisting of tiles. Tiles can be defined as a sequence of two or more k-mers with a fixed overlap length between the k-mers. During error correction, both the k-mer and tile spectrum are used for error correction of a read. Spectrum-based methods

often correct k-mers in a read with their Hamming distance neighbors; a Hamming distance neighbor of a k-mer is defined as the number of positions the two k-mers differ. However, this reduces exactness when an erroneous k-mer has to be corrected since there are multiple candidates for the k-mer. To avoid this scenario, Reptile corrects tiles instead of k-mers. Since a tile has almost twice the character count as the k-mer, error correction at the tile level has far fewer candidates than at the k-mer level. Using the tiles leads to more accuracy in error-correction [81]. The drawback of using both the k-mer and the tile spectrum is that Reptile's algorithm is memory-intensive as it keeps two spectrums in its memory. Please refer to [82] for further details.

2.1.2 Previous Reptile Parallelization Approaches

Previous approaches to parallelize Reptile have either replicated k-mer and tile spectrum on each process or on each node. Both approaches limit the hardware on which parallel Reptile can be run; for example in the work by [70], the spectrums were replicated per process. The approach by Jammula et al. [28] improved upon the original existing parallel Reptile implementation by replicating the spectrum on every node. Since this approach is an improvement over the work by Shah et al. [70], we contrast our approach with this work.

- The k-mer and the tile spectrum were replicated per node compared to the previous approach of replication per process. Multiple threads of each node share the k-mer and tile spectrums during the error correction phase.

- K-mer and tile spectrums are stored as sorted lists with look-up operations involving repeated binary searches over the spectrum. A cache-aware layout of k-mer spectrum was presented which lowered the search time from the original $O(log_2 N)$ to $O(log_{(B+1)} N)$ where B represents the number of elements that can fit into a cache line.

- A dynamic work allocation scheme that depends upon a global master which coordinates the entire work allocation mechanism and a local master that is responsible for getting work from the global master. The actual error correction is performed by

worker threads running on the node who fetch chunks of sequences from the work-queue.

Our approach differs from the the previous work in the following three respects:

- Instead of replicating the entire dataset on a node or a rank, we distribute the k-mer and tile spectrum amongst the processing ranks. This lowers the memory footprint significantly and we have shown that we can perform error correction with a memory footprint of less than 512 MB per process even for the human dataset comprising of over 1.55 billion reads. This implementation also provides highly scalable error-correction times. Thus, our approach provides a memory and time scalable implementation to spectrum based error correction. Besides distributing k-mers and tiles amongst the processing ranks, our implementation also has the ability to replicate the k-mer and tile spectrum on every node. This mode does not require any communication between the processes during error-correction and is designed to be run on machines where the entire spectrum can be replicated on every node. We have also implemented several heuristics which can be employed based on the traits of the datasets and the hardware.

- We store the k-mer and tile spectrum in hash tables instead of arrays; this prevents any need for sorting the arrays or for repeated binary searches. We use separate hash tables for the k-mer and the tile spectrum. The hash tables are created in parallel during the k-mer construction phase.

- We rely on static work allocation for load-balancing. Our approach does not rely on a master-slave policy, instead it redistributes sequences to the processing ranks. We present our results with and without this load balancing policy in Section 3.5, and show that such a static allocation policy balances the load very effectively.

There are several error-correction algorithms and implementations; among k-mer spectrum based methods, besides Reptile, Quake [31] uses a maximum likelihood approach incorporating quality values. For suffix-tree approaches, SHREC [65] was the first implementation to use a generalized suffix trie as the data structure employed for error correction.

Among multiple sequence alignment methods, ECHO [30] performs error correction by finding overlaps between the reads. Several authors have published on parallelization of error-correction approaches; many of the approaches implement shared-memory parallelization only [66] [78] [25]. Only a few implementations exist for distributed memory architectures. This includes Reptile parallelization which we discuss in the next subsection. DecGPU [38] uses both GPUs and distributed memory CPUs to perform error correction.

There has been prior work to use distributed hash tables with messaging to tackle memory-intensive tasks in computational biology such as assembly [2] [7]. A viable alternative to message passing in assembly algorithms is to use a global address space using Unified Parallel C [17].

2.2 Parallelization of Reptile

Related work shows that parallel implementations of Reptile so far have only been run in modes requiring replication of k-mer and tile spectrum. Our approach which differs markedly is detailed below; please note that while most of the changes are detailed with respect to the k-mer spectrum, the same changes also apply to the tile spectrum.

I As a first step, the file consisting of short reads is read in parallel by each rank. The input to parallel Reptile consists of a configuration file, which specifies the fasta file and the quality file to be used for the error correction. At this point, Reptile is not capable of reading the fastq format. The fasta file consists of the sequences along with the sequence names; the names have been pre-processed to be sequence numbers (in ascending order beginning with number 1). The second file to be read is the quality score files, which has information on the quality score associated with every base of the sequence and the sequence number as well. Both files are read in parallel; each rank computes its subset of the reads whose size is simply the file size divided by the number of ranks. The subset of reads are processed beginning with an offset from the start of the file. The offset is based on the rank. Each rank starts reading the fasta file from this offset and records the starting sequence number. It then looks up the same sequence number in the quality score file as well to ensure that the quality scores

corresponding to the same set of reads as the fasta file is processed. Similarly, the end sequence number is also computed. Each rank is responsible for the set of reads corresponding to the starting sequence number up to the ending sequence number. This subset of reads is read in chunks by each rank; the chunk size is also defined in the configuration file.

II In this step, each rank builds a k-mer and a tile spectrum from its set of reads. The k-mer spectrum is represented by key-value pairs with k-mer ID as the key and the count of the k-mer as the value. The k-mer ID is a number constructed from the characters of the sequence. The tile spectrum is similarly represented except the tile ID is a long integer as the number of characters of the tile can be up to $2k$ where k represents the number of characters of the k-mer. The k-mer and tile spectrum are stored in separate hash tables on each rank. With each read, the k-mers and tiles corresponding to the reads are processed, and added to the k-mer and tile hash tables respectively.

During the current step, the k-mer and tile spectrum are separated into the *hashKmer* and *readsKmer* hash tables, and the *hashTile* and the *readsTile* hash tables respectively. Each k-mer (and tile) are defined to have an owning rank; the owning rank in our implementation is defined as the rank p (out of the number of ranks np) for which $hashFunction(kmer)\%np == p$ (and similarly for the tile). The rank adds the k-mer it has processed to the *hashKmer* if it is the owning rank, else the k-mer is inserted into the *readsKmer* hash table. The process continues until the entire allocated subset of reads are processed by the rank. Once this phase is over, the rank stores the k-mers extracted from its reads in either a hash table consisting of the k-mers it owns (*hashKmer*) or a hash table consisting of the k-mers (*readsKmer*) it does not own (and similarly for the tiles). As can be observed, this is an embarrassingly parallel phase requiring no communications amongst the ranks.

III After the previous phase, each rank has two hash tables, each for the k-mer and tile

spectrum. However, it can be observed that no rank has the true global counts for the k-mers and the tiles in their hash tables. This is because each k-mer might exist on multiple ranks (as part of their *readsKmer* hash table), besides the owning rank. The counts of the k-mer on the owning rank thus need to be added to the counts of the same k-mer that exist on every other rank to get the true global count of the k-mer. This next phase thus requires communication such that all k-mers and tiles (along with their true global counts) exist on only the owning rank. For this step, each rank processes each k-mer in its *readsKmer* hash table; for each k-mer, the owning rank is computed ($hashFunction(kmer)\%np$) and the k-mer and its local count is placed into a vector for the owning rank. This is then followed by an *MPI_alltoallv* communication that sends a vector of k-mers and their counts to their owning ranks. Each rank then processes the k-mers it has received from the other ranks. This step involves adding to the count of the k-mer if the k-mer exists in the hash table, or adding the k-mer (along with its count) if it does not exist. Following this, each rank now has a hash table of k-mers it owns, with the true global counts (or frequencies) of these k-mers.

Finally, based on the threshold set in the configuration file, k-mers and tiles below a threshold are subsequently removed from their hash tables by the ranks. A memory-efficient alternative to this step is usage of a Bloom filter [17]. Note that the with these steps, each rank only retains now a subset of the k-mer and tile spectrum with their true global counts; the storage requirements of each rank for the k-mer (and tile spectrum) now depend on the hashing function. With the inbuilt hashing function of the C++ standard templates library, we have found the number of k-mers and tiles to be remarkably consistent across the total number of ranks. The total time taken up by the the steps I-III is printed as the k-mer construction time in our execution; besides the error correction times, we also show the k-mer construction times in Section 3.5. Figure 2 shows the steps of the k-mer construction including parallel reading, construction of the hash and the reads k-mer tables followed by the collective communications for a hypothetical execution of 128 ranks and k-mer size of 3. These steps are similarly executed for tiles as well.

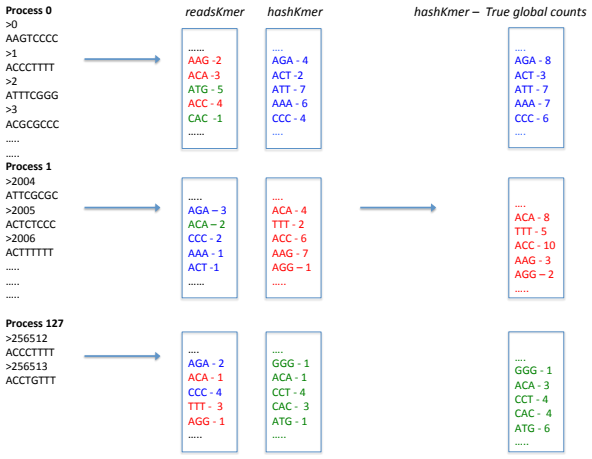

Figure 1: A diagram representing the steps of the k-mer construction for a hypothetical execution of 128 ranks and a k-mer size of 3

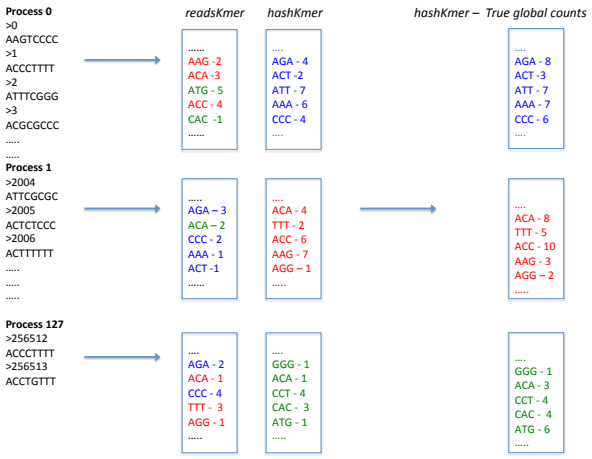

Figure 2: A diagram representing the steps of the k-mer construction for a hypothetical execution of 128 ranks and a k-mer size of 3

IV After the k-mer construction steps, each rank now has a hash table of the k-mers and tiles it owns (and an optional hash table of k-mers and tiles that it has processed from its reads dataset). Once all the ranks have finished the two steps, the error correction step can now begin. For our experiments, the short reads are again processed from the file. This is because the total memory consumed by storing the reads will increase the memory footprint significantly. Most of our experiments are run with only 512 MB per

rank, thus storing the reads is not a feasible option for us. The error correction of each read requires a set of k-mers and tiles (and their Hamming distance neighbors). Each rank at the beginning of this step forks two separate threads - one thread is responsible for the error correction of the reads in its part of the file, while the other thread acts as a communication thread. If a rank during error correction does not have a k-mer (or tile), it first finds out if it is the owning rank. In case the processing rank p is the owning rank, this implies that the k-mer or tile does not exist; in case the processing rank is not the owning rank, it looks up its *readsKmer* hash table (in case of the corresponding mode of execution). If the k-mer is not found, it sends a message to the owning rank, requesting the count of the k-mer or tile. The communication thread of each rank probes any incoming messages – based on the probe, it first finds out the nature of the request (if it a k-mer or a tile lookup). The thread then looks up the corresponding hash table (k-mer or tile) based on the request) and sends the appropriate response. The response is either the count of the k-mer or tile or a response like (-1) implying that the k-mer or tile does not exist. If a k-mer or tile does not exist at its owning rank, it can be inferred that the k-mer or tile does not exist at all in the entire k-mer spectrum.

Each rank can continue with the error correction of its reads subset using the strategy above; the communication thread of the rank responds to incoming requests, while the non-communication thread continues with the error correction of the rank's subset of reads using Reptile's algorithm. Once all the ranks have finished their error correction step, each rank shuts down its communication threads and outputs the reads it has corrected.

2.2.1 Load Balance Through Randomization

One issue we faced with the strategy above is load imbalance which Jammula et al. [28] have also explained in their work. This issue is because in many cases, the errors appear localized in several parts of the file. Since the reads in the file are divided up into chunks amongst the ranks, this leads to certain ranks having considerably more erroneous sequences compared

to the other ranks. Since the work done with erroneous sequences is much higher than the other ranks, this leads to load imbalance between the ranks. In Section 3.5 we show the variation in times between the slowest and the fastest ranks, along with a breakdown of their times for 128 ranks processing the E.Coli dataset. While Jammula et al. [28] have relied on a dynamic load balance approach based on a global master, a local master and worker threads, our approach of load balance is a static scheme. As we noticed that most of the load imbalance is caused due to errors being localized in parts of the file, a "randomization" of the entire file might remedy the problem.

Our strategy is for a sequence to be processed by a rank only if it is the owning rank; a sequence is designated to be owned by a rank p if $hashFunction(seq)\%np == p$ (similar to our definition for k-mers and tiles). Therefore, in addition to the Step I above, we also perform additional steps for load-balancing: after each rank has read their batch of short reads (or sequences), they find out the owning rank for each sequence in their batch of reads. The sequences are then placed in separate buckets corresponding to the owning ranks. Subsequently, a collective communication *MPI_Alltoallv* is performed; each rank then processes the sequences for which they are the owning rank. This hashing of sequences has the same effect as the "randomization" of the file might have.

2.2.2 Heuristics for Efficient Parallelization

Besides the core steps above, we have also implemented heuristics to be employed for efficient execution based on the dataset and the architecture. The primary purpose of these heuristics is to lower the runtime or memory footprint based on the hardware being tested and the requirements of the dataset. We show the results for all the heuristics in Section 3.5 for the E.Coli dataset for 32 nodes only. Since we had limited access to higher number of nodes, we are unable to present the results of the heuristics for higher node counts and other datasets. However, the results for E.Coli give us an understanding of the effectiveness of these heuristics. We give a brief description of each the heuristics below.

- **Universal**: We rely on message passing between the rank's communication threads to get the counts of k-mers and tiles. Based on the request (k-mer or tile), the sending

rank puts different tags on the messages. The receiving rank probes any incoming messages (with any sending rank or any tag) and subsequently based on the tag, looks up the hash table corresponding to either the k-mers or the tiles. In *universal* mode of execution, the message is itself a structure with the tag included as part of the message (and the k-mer ID and the tile ID). The receiving rank now does not have to probe the message for the tag, but accepts any message; once the message is received, it looks up the tag (which is part of the message received) to find the nature of the request and subsequently does a lookup of its hash tables. This increases the size of the message but makes the call to MPI_Probe unwarranted.

- **Read K-mers/Tiles**: Each rank retains the k-mers it owns. Besides the owned k-mers, it can also have k-mers and tiles from its own set of reads. To implement this heuristic, an additional collective communication step is needed where each rank sends the k-mers it does not own to the owning rank, requesting the global count for the k-mer. This is again implemented as an *MPI_alltoallv*, where each rank creates a vector of k-mers to be sent to the owning ranks. Thus, in this mode, each rank has two hash tables each for k-mer and tiles (or four in total). If a k-mer is not found in the *hashKmer*, it is looked up in the *readsKmer* and then a message is sent to the owning rank. This increases the local lookup time, but can potentially decrease the time spent in communication.

- **Allgather k-mers/tiles/both**: This heuristic replicates the entire k-mer spectrum and/or tile spectrum on every node. This mode is designed to be used on machines with enough memory to keep either or both the spectrums. This mode does not employ any message passing between the ranks during the error-correction step. The only communication in the entire execution is the collective communication calls to exchange the owned k-mers and tiles amongst the ranks. As expected, this mode decreases the runtime significantly, while increasing the memory footprint.

- **Add remote k-mer/tile lookups**: This mode adds any k-mer or tile lookups from the remote ranks; once a k-mer or tile count is received from the owning rank, it adds

those k-mers to the local *readsKmer* hash table. This mode can only be run with the *read kmers* mode as the remote k-mers and tiles are added to the *readsKmer* and the *readsTile* hash table. The lookup strategy follows the *read kmers* heuristic; a k-mer is first looked up in the owned k-mers hash table *hashKmer*, followed by the reads hash table *readsKmer* and finally requested from the owning rank. This mode will be useful if the k-mers or tiles needed from remote ranks, will be needed in the future.

- **Batch Reads Table**: In this mode, the reads table which is maintained during the k-mer construction phase (and optionally during the error correction phase), is kept to a minimum size with an increased communication overhead. Each rank reads a chunk of their subset of reads (specified by the chunk size in the configuration file). This mode performs Step III of the parallel algorithm after each batch of reads instead of performing it in the end once all the ranks have processed their set of reads. In this mode, after all the processes have read their batches, the processes synchronize and complete a *MPI_alltoallv* operation to assign the k-mers and tiles to their owning ranks from the batch of reads just processed. Each rank subsequently processes the set of k-mers and tiles it has received and adds them to their hash tables for the k-mers and the tiles. Following this, the reads hash table is emptied out before the next batch is read. Thus, the size of the reads hash table can be kept to a minimum by varying the chunk size as the reads hash tables only contain k-mers and tiles from a single chunk than from all the chunks. One point of observation is that different processes might have been allocated slightly different number of batches; thus, before this step, a *MPI_Reduce* is carried out to find the maximum number of batches amongst all the processes. Each process thus continues this process for the maximum number of batches even though it might have exhausted its set of reads. This is because other ranks might still be continuing to process their set of reads and require every rank participation in the *MPI_alltoallv* operation. This mode lowers the memory consumed with an increase in the communication overhead. However, since the k-mer and tile construction time is a negligible percentage of the total time for error correction, the overhead is not substantial.

Table 1: E.Coli, Drosophila and Human datasets used for experimentation

Genome	Number of reads (millions)	Length (chars)	Genome Size	Read Coverage
E.Coli	8874761	102	$4.6 * 10^6$	96X
Drosophila	95674872	96	$1.22 * 10^8$	75X
Human	1549111800	102	$3.3 * 10^9$	47X

2.3 Results

In this Section, we discuss the results of our implementation for 3 datasets - E.Coli, Drosophila and human dataset. We have followed exactly the same methodology of preparing the datasets as [28]; our datasets are very similar to Jammula et al. with minor differences being introduced in the conversion of the downloaded fastq file format to separate fasta and quality score files which are needed by Reptile. Table 1 shows the details for the datasets including the number of the reads and the length of the reads. The smallest dataset is the E.Coli dataset with less than 9 million reads, with the human dataset roughly 1500 times the size of the E.Coli dataset. The read coverages for all the genomes have been calculated as (Length*Number of Reads)/(Genome Size)

We have tested our implementation on IBM's BlueGene/Q architecture. Blue Gene/Q (BG/Q) is the third generation of highly scalable, power efficient supercomputers in the IBM BlueGene line, following Blue Gene/L and Blue Gene/P. The BG/Q SoC has 16 cores for user code, and a 17th core is reserved for use by the system software. Each core has four hardware threads. The 4 threads are simultaneously multi-threaded (SMT) threads. Each node has a wake-up unit that allows SMT threads to "sleep" waiting for an event; this allows faster OpenMP work handoff and lowers messaging latency. The 64-bit, in-order, PowerPC cores run at 1.6 GHz. 32 compute nodes are electrically interconnected to form a 2x2x2x2x2 grid on a node card. 16 node cards comprise a 512-node midplane and two midplanes stack vertically to form a 1024-node rack, with electrical links within midplanes and optical links between midplanes.

Our BlueGene/Q hardware has 16 GB of memory per node. For most of our experiments,

we have decided to run 32 ranks per node, with each rank running 2 threads (communication and error correction) during the error correction phase. During error correction for most of our experiments, we run 64 threads per node which is the maximum possible number of threads on the BG/Q node. Using 32 ranks only allows each rank to have 512 MB per process; this includes memory for both the messaging buffers and the application's data structures. Using multiple ranks per node also gives us a benefit: it allows any communication between the ranks on the same node to use the shared memory on the node (and not use the messaging interface). Considering the nature of the problem, most of our effort has been to lower memory footprint per process. We have not made an extra effort to optimize the communication between the ranks; this has been done purposefully to observe if we can get good results without any tuning that may be specific to the datasets.

To find the effect of running multiple ranks per node on runtime, we varied the number of ranks per node from 8 to 32. Figure 3 shows the results for the E.coli dataset using 128 ranks for this run; the number of ranks per node are varied from 8 to 32. Thus, the number of nodes are varied from 16 to 4 for this experiment. As can be seen from Figure 3, the time taken using 32 processes per node is slower than using 8 or 16 processes per node by almost 30%. Most of the increase comes from slowdown in communication. This slowdown is expected as each node of BlueGene/Q only has 16 physical cores per node; with 8 processes per rank and 2 threads per rank, the cores are fully occupied. Increasing the number of threads beyond 16 per node leads to usage of the hardware threads. The setting for least run time is 8 ranks per node; however our primary focus is to minimize the number of nodes (and not necessarily the runtime) for execution, and all our experiments for the E.Coli, Drosophilla and the human datasets are run with 32 ranks per node. The runtimes will further reduce if the number of processes are reduced to 8 or 16 per node for our experiments.

Figure 3 also provided some observations about the application overall; looking at the times in k-mer construction and error correction, it can be seen that the k-mer construction time is a negligible percentage of the error correction time. Most of the error-correction time is spent in communication as expected; it can also be observed that the majority of

the communication time is spent in communication of tiles especially tiles which are not part of the tile spectrum (non-existent on any rank).

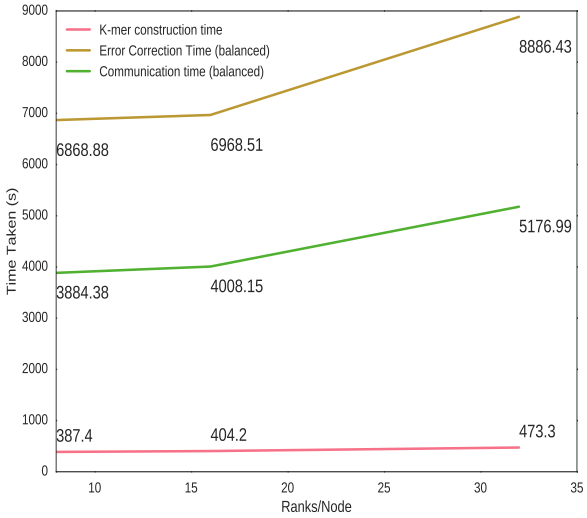

Figure 3: Execution time of 128 ranks for the E.Coli dataset varying the number of nodes from 4 to 16 nodes.

For our implementation, it is key to keep the memory footprint of the processes uniform. This implies no particular rank has a substantially higher count of k-mers and/or tiles that could become a bottleneck; if a rank has a much higher count of k-mers or tiles, that could lead to more messaging overhead for that rank during error correction. Figure 4 shows the total number of the k-mer and tiles for all 128 ranks of the E.Coli dataset. For this run, the variation between the ranks having the highest and the lowest number of k-mers is less than 1%, with the variation in the number of tiles slightly less than 2%. This shows that the distribution of the k-mers and tiles is uniform across all the ranks, making the memory footprint and the messaging overhead of each rank consistent.

Figure 5 shows the effect of load balance on our results for the E.Coli dataset for 128 ranks on 4 nodes; our static load balancing algorithm reduces the total runtime almost by a factor of 2. Without load balance, there is a huge variation in the number of errors corrected per rank. The lowest number of errors corrected amongst all the ranks is 33886, while the highest number of errors corrected are almost 50% higher to 47927. This leads to the load imbalance; the fastest rank in this case takes 4948 seconds, while the slowest rank

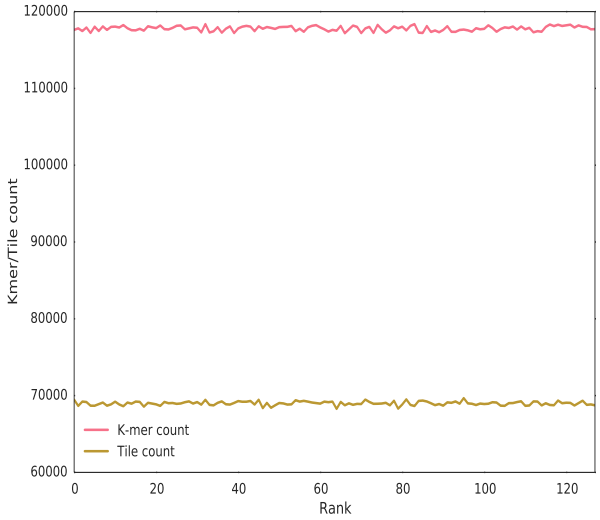

Figure 4: K-mer and tile count of each rank for 128 processes

takes more than triple that time to more than 16000 seconds. The majority of the time is taken up in the communication time varying from 2891 seconds for the fastest rank to more than 10800 seconds for the slowest rank. Most of the communication time is taken up in the communication for tiles. The breakdown of the communication time shows that while the fastest rank needs almost 31 million remote tile lookups, the slowest rank needs more than 118 million tile lookups (not shown in Figure 5).

The load balancing strategy makes a major difference in the results; almost all the ranks uniformly take 8886 seconds. The number of errors corrected per rank only vary from 39127 to 39997 (only 2%), with the range of the communication time from 5073 seconds to 5268 seconds (less than 4%). The remote tile lookups needed per rank stays remarkably consistent with 64 million lookups per rank. Since there is a considerable improvement with load balancing, all of our future experiments are also completed with static load balancing.

As detailed in Section 3.3, our implementation also has support for various heuristics for optimal execution. For all the heuristics, we only show results for 1024 ranks running on 32 nodes for the E.Coli dataset. It is possible that other datasets might show different results for the heuristics, but since we had limited access to the BlueGene/Q for node counts higher than 32, we only experimented with these heuristics for the E.Coli dataset.

Figure 6 shows the results for all the heuristics in terms of time taken and the highest

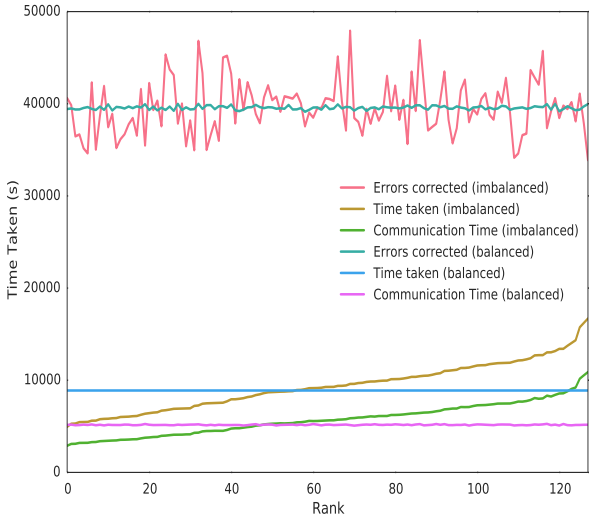

Figure 5: Time taken (total and communication) and errors corrected for 128 processes

memory footprint rank after the k-mer construction and the error correction steps. We can make several observations from Figure 6 about the effect of the heuristics on the runtime and the memory footprint. We detail the observations of each heuristic below:

- Universal mode is faster than non-universal mode by 8.8%. The increase in performance doesn't consume any extra memory, and thus this mode is advantageous to the non-universal mode.

- Replicating the k-mer spectrum on every process leads to a deterioration in performance; these runs were completed with 8 ranks per node (or 256 total ranks) as the memory footprint was noticeably higher. Due to the lower number of ranks, the improvement by replicating the k-mer spectrum on every rank is offset by the increased workload of the ranks. The memory footprint increases to 928 MB per rank as well.

- Replicating the tile spectrum on every process reduces the runtime of the error-correction step to 975 seconds (from 1178 seconds of the base mode). With the replication of tile spectrum, no communication is needed for the tiles; since the runtime is dominated by the communication time of tiles, the runtime decreases even with the lower number of ranks. The replication also increases the memory footprint to 948 MB per rank. Thus, instead of replicating both the k-mer and the tile spectrum

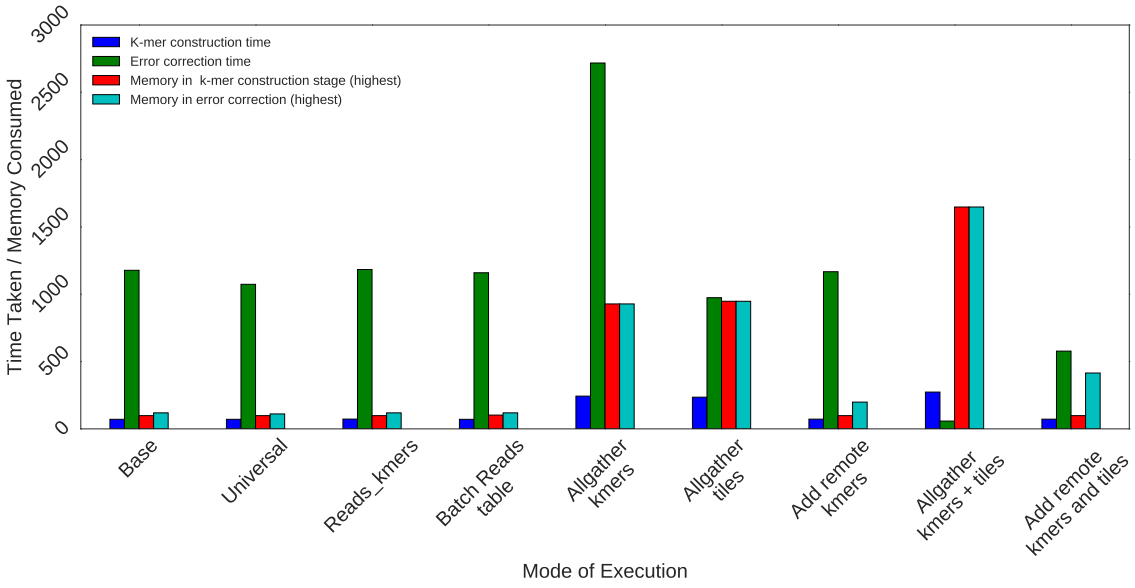

Figure 6: Time of execution and memory footprint with different heuristics

on every rank, it is highly advantageous to replicate only the tile spectrum, relying on communication for the k-mer spectrum. This run was also completed with 256 ranks only.

- The effect of adding remote k-mers (both the ones which are on other nodes and the ones which are non-existent) does not improve the runtime of the error-correction step. The memory footprint increases to 199 MB from 119 MB for this heuristic.

- Batch reads table is useful in lowering the memory footprint further by keeping the size of the reads hash table to a minimum. This run was completed with a chunk size of 2000 reads; the reads table is processed and cleared after each rank has finished their chunk of reads. Since the number of sequences processed by each rank is 8657, each rank does this step 5 times. This heuristic is highly advantageous as it was used for the runs for the human dataset.

- Adding the k-mers and tiles belonging to the reads of the rank does not improve the performance of the error correction step. This is because most of the communication time is spent in remote tile lookups.

- Finally, with the k-mers and tiles replicated on every node, the error-correction time is only 58 seconds. The memory footprint of this mode is almost 1648 MB per rank. This run was completed with only 1 rank per node and 64 threads per rank.

For our purposes, the advantageous heuristics are *universal* which reduces the runtime, and *batch reads table* which reduces the memory footprint of collective communications. We did not use any of the replication settings in our experiments.

Figure 7 shows the results for the E.Coli dataset as the number of ranks are increased from 1024 to 8192; since each node is running 32 ranks, this translates into an increase in the number of nodes from 32 to 256. No heuristics were employed in this scalability graph. Figure 7 shows that our implementation is scalable both in k-mer construction and error correction times. The parallel efficiency for E.Coli dataset at 8192 ranks when the error-correction time is approximately 180 seconds is 0.81. Figure 7 also shows the improvement over imbalanced run; there is a marked improvement in runtimes especially at lower node counts. For example for 32 nodes, the runtime more than halves due to our strategy of redistribution of sequences for load balance. At 256 nodes, the total time taken to correct the dataset is less than 200 seconds using the load-balancing approach.

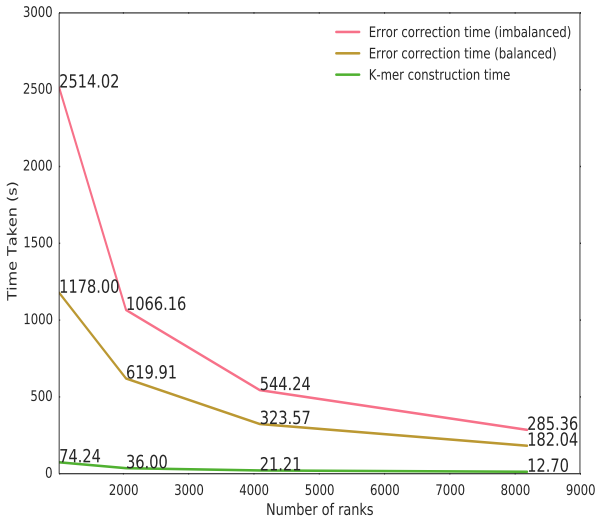

Figure 7: Scaling graphs for E.Coli dataset varying the number of nodes from 32 to 256

Figure 8 shows similar scalability results for the Drosophila dataset as the number of

ranks are increased from 128 to 512 BlueGene/Q nodes. This figure also shows excellent scalability from 1024 ranks to 8192 ranks. In this graph as well, each node is running 32 ranks per node. The load balancing improves the performance significantly; the runtime improves by more than a factor of seven at 8192 ranks (or 256 nodes). The runs using the imbalanced approach for node counts 1024 and 2048 did not finish in a reasonable time. Also, it can be seen that for 1024 ranks, the K-mer construction time takes 981 seconds. This run was completed with the heuristic *batch reads table*, which reduces the memory footprint of the k-mer construction stage, but leads to an increase in runtime. The parallel efficiency at 8192 ranks for Drosophilla dataset is 0.64.

Figure 8: Scaling graphs for Drosophila dataset varying the number of nodes from 128 to 512

Finally, Figure 9 shows the runtimes for the human dataset consisting of over 1.55 billion reads varying the number of nodes from 128 to 1024. The runs were completed with 32 ranks per node, so the total number of ranks are varied from 4096 to 32768. All the runs were completed with the heuristic *batch_reads* and the load balancing strategy enabled; the reason for the *batch_reads* heuristic being employed is the size of communication buffers for the collective communication in Step II of the k-mer construction time exceeds the memory available on each process. As explained before, this heuristic leads to multiple collective communication calls in the k-mer construction; each call is executed after all the processes

have processed a batch of reads. For the 128 and the 256 nodes run, the batch size was only set to 5000 reads, while for the 512 and 1024 node runs, the batch size was set to 10000 reads. A BlueGene/Q specific environmental flag to lower the memory requirements for collective communication by implementing the collective calls as multiple point to point communication was also used. **This result shows we can complete error correction of the human dataset in less than 2.5 hours using a single rack of BlueGene/Q.**

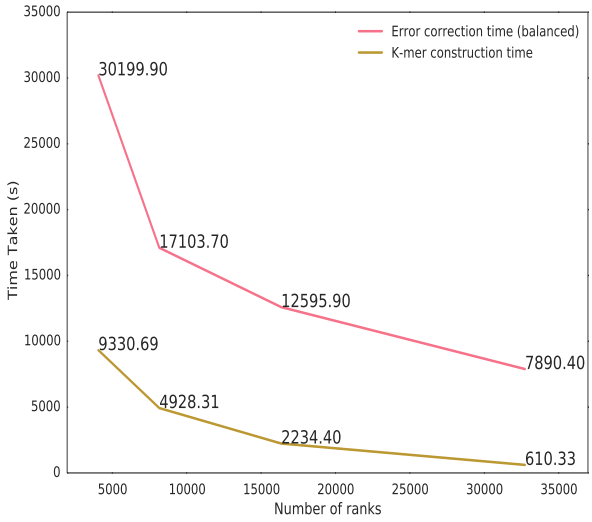

Figure 9: Scaling graphs for human dataset varying the number of nodes from 256 to 1024

2.4 Applying Collective Communication in Previous Algorithm

In the previous algorithm explained in Section 3.3, each process keeps a subset of the k-mer and tile spectrum, relying on point to point message passing amongst the processes to get the counts for the k-mers and tiles it does not "own". While this algorithm scales on our target architecture BlueGene/Q, we have also implemented another algorithm that generates the tiles beforehand and uses collective communication before using point to point communication. The algorithm only changes the error-correction step still relying on the previous steps of generation of distributed k-mer and tile spectrum. This algorithm for error-correction using collective communication is described in the steps below:

- The error-correction step starts with each rank processing it's batch of reads. With this algorithm, each process first generates the tiles required by the batch of sequences

Table 2: Comparison of implementations for E.Coli dataset

Implem.	Total Time	Comm. Time	Tile Generation Time	Tile Hashing	% Collective k-mer lookups	% Collective tile lookups
Collective	1589.02	111.24	1086.23	674.1	11.26	72.78
Non-collective	1188.45	569.1	0	0	0	0

it has read. The tiles generation of a sequence depends on whether it will be error-corrected or not – this in turn depends on several heuristics. Each process keeps the tiles it needs the counts of and the tiles it doesn't own in a separate hash table. This hash table is regenerated for every batch of reads. Once these tiles are generated for all set of reads, the next step places the tiles into separate bins depending on which process "owns" them. For each tile, the owning rank is computed $(hashFunction(tile)\%np)$ and the tile is sent to the owning rank for it's count. Each owning rank then processes the tiles it has received from the other ranks, sending their counts to the ranks who have requested the tile counts. This step is also implemented as a collective communication *MPI_Alltoallv* call. The tile counts are then updated in the hash table. Since the tile generation and lookup depends on the k-mers associated as well, the k-mer counts are also kept in a separate hash table which is also generated with every batch of reads.

- The next step is the error-correction phase. With the previous step, the error correction step of finding a k-mer and tile is changed slightly. First, the hash table of the generated tiles and k-mers are looked up, followed by the original methodology of looking up a k-mer in it's own hash table (incase it is the owning rank) or the remote "owner" hash table of the k-mer.

  With these steps, the communication time especially for the tiles is reduced significantly. However, in our current experiments on BlueGene/Q, the time to generate the tiles is significant which leads to an overall slowdown compared to the previous strategy of using only point to point communications.

Table 3: Comparison of implementations for E.Coli dataset on Power8

Implem.	Total Time	Comm. Time
Collective	12804.2	10597.4
Non-collective	24170	22568.7

Table 2 shows the difference in timings between the original and the collective implementation using 1024 nodes of the BlueGene/Q. This shows that overall collective implementation is slower taking a total of 1589.02 seconds vs. 1188.45 seconds. For the collective communication time, the communication time however is much lower at 111.24 seconds versus 569.1 seconds. However, the generation of tiles takes up a significant percentage of the total time with 1086 seconds of the total time of 1589.02 seconds. 72.78% of the total tile lookups and 11.26% of the total k-mer lookups are completed using collective communication before the error-correction of a batch begins. For these runs, the batch size was 1000 sequences only as the number of potential tiles per batch of reads is very high. At this time, the BlueGene/Q we had access to at Yorktown Heights has gone offline, thus we are unable to continue with our experiments using this updated algorithm on BlueGene/Q. We also benchmarked the performance of the collective and non-collective implementation on IBM's Power8 cluster at Yorktown Heights. We ran the EColi dataset on 4 nodes of the Power8 processor; each node comprised 2 Power8 processors for a total of 20 cores. Total of 80 processes were launched each with 2 threads; Table 3 shows the comparison of timings with collective and non-collective modes. As can be seen from Table 3, collective communication based approach is about twice faster than the non-collective approach. With non-collective approach, the majority of the time was being taken by communication of tiles. However now with the collective approach, more than 70% of the time was now infact taken up by the communication of k-mers. We were not able to continue our experiments on Power8 as our access to Power8 was discontinued.

2.5 Using Distributed K-mer Spectra for Metagenomics

As explained before, the first step of implementation of parallel Reptile was to construct a distributed k-mer spectrum. We are currently reusing our implementation of distributed k-mer spectrum for problems in metagenomics as well. The first step of this approach was to create a library API with distributed k-mer spectrum. Our library API is *processKmers-FromReads(filename, batchsize, kLength, rank, size, hashKmer, write_to_file)*; here is a description of each of the arguments:

- *filename* is the input reads file.
- *batchSize* is the number of sequences read at one time from the reads input file before collective communication is completed to exchange the "owned" k-mers.
- *kLength* is the k-mer length.
- *rank* is the MPI rank of the process.
- *size* is the total number of MPI processes.
- *hashKmer* is a custom defined hash table (using C++'s *unordered_map* where both the key and value pairs are 64-bit unsigned integers.
- *write_to_file* is a boolean flag which writes out a combined k-mer file once it is computed. Each k-mer is stored as a 64-bit unsigned integer which is constructed from the characters of the reads file.

The library includes functionality to accomplish several of the tasks needed for distributed k-mer spectrum such as parallel reading of a FASTA file, exchanging "owned" k-mers, computing the number of batches for the entire file reading etc, but these are hidden from the user for ease of use.

Since the parallel Reptile work, we have also removed the constraint of the sequence names to be numbers only. With this change, we can now handle parallel reading of arbitrary FASTA files. As a first step, each process computes an offset and the number of bytes of it's subset based on it's rank, total number of ranks and the file size. With the offset, each

process computes a starting sequence name s from where it begins reading the file. Each process starts reading it's subset of the file beginning with it's offset. Once a rank p has exceeded the number of bytes it has to read b, it registers the ending sequence name e as well – the rank p is thus responsible for reading the sequences from the starting sequence s to e, not including the ending sequence. Each process p from *0* to *n-1* (n is the total number of processes) now sends it's ending sequence name e to the process *p+1* which now makes it's starting sequence name as the ending sequence name it has recieved from the process p. This process thus sets up the starting and ending sequence name of each process; after this process, each process subsequently reads the entire subset of it's file in batches. The number of batches is dependent on the batch size *batchSize*. Once the distributed k-mer spectrum is computed, a combined file with all the k-mers and their counts is also written to disk. This filename is generated with both the input filename and the k-mer length to distinguish the file. Each process writes it's own separate file to disk; rank 0 subsequently combines all files into a single file containing the entire k-mer spectrum with it's counts. Other efficient methods of file writing are also being currently implemented. Other k-mer based methods in the library include functionality of reading from this k-mer file: *processKmersFromFile* supports efficient parallel reading of k-mers from this file similar to *processKmersFromReads*. Other utility programs have also been added to the library such as processing k-mers of an entire directory of reads files in FASTA format; this program also has support for checkpointing such as it only processes the reads files for which k-mer files do not already exist on disk.

2.5.1 Using Bray-Curtis measure for metagenomics

Previous research [13] has focused on k-mer spectrum analysis for ascertaining metagenomic dissimilarity. Shotgun metagenomics is the analysis of extremely large and fragmented sets of DNA sequences from a complex microbial community, present in all natural environments. This technique generates millions to billions of reads from the total DNA of the genomes of all organisms inhabiting the environment. To fully analyze these reads, the reads are either *de novo* assembled or aligned to known species. The alignment method compares the reads

files against whole genomes of known organisms; this method is time-consuming and has usefulness only if there is a prior knowledge of the reference genomes that are in the sample. *De novo* assembly on the other hand assembles the reads files and is usually a daunting task for metagenomics considering the existence of unknown genomes with varying abundance.

Challenges in both approaches has to led to research into k-mer spectrum based methods for metagenomic dissimilarity analysis [13]. This approach uses pairwise distance computed on the basis of k-mer vector (k-mers and their normalized counts) to ascertain the similarity between two metagenomic samples. The normalized Bray-Curtis measure for two metagenomic samples x and y is defined as follows:

$$BC_n(x,y) = 1 - 2\sum_{i=1}^{i=4^k}(min(m_i(x), m_i(y)/\sum_{i=1}^{i=4^k}(m_i(x) + m_i(y))$$

where $m_i(x)$ denotes the normalized count of k-mer i. For distributed k-mer spectra, the k-mer is "owned" by rank p if $hashFunction(kmer)\%np == p$. For two metagenomes x and y, if the hashing function is the same, the k-mers common to both metagenomes will be "owned" by the same rank p. The Bray-Curtis sum $\sum_{i=1}^{i=4^k}(min(m_i(x), m_i(y)$ can thus be computed on a per-wise rank basis, and then summed up across all the ranks. The only communication is needed in the final step when a *MPI_Reduce* is used to sum the per-rank measure on rank 0. Thus, the most important step of the Bray-Curtis measure is the efficient computation of distributed k-mer spectra. Previous approaches have used Bray-Curtis measure to ascertain dissimilarity between metagenomic samples; however their approach was completed with a few constraints.

- K-length (k=15) was used in their study because of limitations of memory per node. Since the study was completed on a single node, the authors were constrained to keep the k-size small to keep the entire k-mer spectrum in memory. With our distributed k-mer spectra computation, we are not limited by this constraint increasing the k-length to upto 30 to find out if higher k-lengths give a bigger insight into the metagenomic samples.

- The authors due to limitation of their implementation used only a single k-length; this

is because doing an all-to-all comparisons using multiple k-lenths would be computationall prohibitive. Since our k-mer spectrum computation is almost linearly scalable, we can vary k-lengths from 15-30 to find out the variability of the dissimilarity measure with varying k-lengths.

- The authors also used sampling of the reads files as the reads datasets exceeded millions of reads. Again, since our approach is scalable, we can do Bray-Curtis measure for the entire reads files and not just the samples.

At the time of writing this document, further research [19] has continued using the parallel distributed k-mer implementation for metagenomic analysis. Figuure 10 shows the comparisons of k-mer and reference based approaches for metagenomic analysis using 100 metagenomic samples simulated using MetaSim[60]. This shows a correlation between the two approaches, and shows that k-mer spectrum based approaches might offer alternatives to reference based methods.

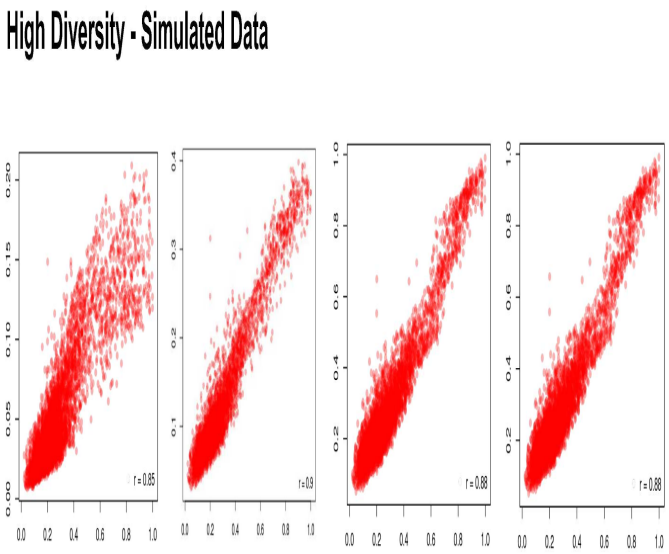

Figure 10: Comparisons of k-mer and reference based approaches for metagenomic analysis. Left to right, the figures show comparisons with k-mer lengths of 7, 11, 21 and 31

CHAPTER III

PARALLELIZATION OF THE TRINITY PIPELINE FOR *DE NOVO* TRANSCRIPTOME ASSEMBLY

3.1 Introduction

High throughput sequencing technologies are making a big impact in many areas of life sciences. The most well-known technique is genome sequencing, whereby an organism's whole complement of DNA is sequenced. Reference genomes are being generated for an increasing number of organisms, including some extinct species [56]. Variation within a species is now being considered [16], including hypervariation in cancers [64]. Nevertheless, sequencing is applied in many other areas of life sciences including transcriptomics, ChIP-Seq, metagenomics, etc.

In this work, we have focussed on transcriptomics in which a library of cDNA derived from a sample of RNA is sequenced. The aim is to sequence the mRNA corresponding to transcribed genes, and various techniques exist to select for mRNA or remove other kinds of RNA (such as ribosomal). Transcriptomic sequencing differs from genome sequencing in two crucial ways. Firstly, the population of mRNA depends on the expression levels of genes in the chosen sample, and there can be a very large dynamic range. Secondly, in higher organisms genes are post-processed by alternative splicing to generate multiple isoforms, which need to be distinguished.

Transcriptomics gives information on which genes are expressed in a given cell type, under given conditions, at a given time point. The number of reads which map to a given gene or isoform is a direct measure of the expression level. Thus, transcriptomics is a major route to the study of gene expression, and is rapidly replacing microarrays as the method of choice.

As in genome sequencing, the cDNA libraries are chopped up into millions of short reads which are then sequenced. The computational task is then to re-assemble these sequenced

reads into a set of transcripts corresponding to gene products. As the sequencing technology improves, this computational step is becoming the principal bottleneck. A dataset consists of a large set of sequenced reads (provided as a FASTA or FASTQ file) which can have a size on the same order as for genome sequencing. Unlike genome sequencing though, an organism can have multiple transcriptomes corresponding to different cell types or conditions. The recommended practice is to sequence these together into a consensus transcriptome, and thus the size of the dataset is multiplied by the number of experiments considered.

Transcriptome assembly thus works on very large datasets and can require considerable compute resources. It is a high-performance computing (HPC) problem. Nevertheless, many of the commonly-used bioinformatics tools were developed for desktop applications, and most assume a shared memory architecture. Such software struggles with today's large datasets, taking days to run routine jobs, and often exceeding the available memory. At the same time, the trend for HPC is towards more parallelism, with larger numbers of lower power nodes [29]. Adapting bioinformatics tools for multi-node distributed memory architectures is thus essential. With a few notable exceptions [72][27][3], distributed memory architectures are not well supported by existing bioinformatics tools.

We have chosen to look at the Trinity pipeline [18] which is one of the most popular packages for transcriptomic assembly [76]. It was originally created for *de novo* assembly, although there is now a protocol for assembly against a reference genome. *De novo* assembly constructs the transcriptome solely from the available reads, and is useful when there is no reference genome available, or if there are likely to be large structural variations (e.g. in cancer cell lines) [43].

Trinity is a pipeline implemented in Perl which wraps a number of underlying programs implementing different stages of the assembly [26]. In it's evaluation, Trinity has been found to be accurate in transcriptome assembly under a variety of conditions, but with high runtimes [85]. The pipeline is very heterogenous in its computational requirements, with early stages requiring large amounts of memory, and later stages being more CPU-intensive. Trinity's assembly pipeline consists of four consecutive modules: Jellyfish,

Inchworm, Chrysalis and Butterfly. The modules are separate executables, written in different languages. The Chrysalis step itself is composed of separate submodules including Bowtie [32], GraphFromFasta and ReadsToTranscripts. Previous attempts to speed up Trinity have focused on using OpenMP threads in a shared-memory architecture and reducing I/O operations [23]. We present results here for the Chrysalis module, aimed at speeding up this section of the pipeline by spreading the load across multiple distributed nodes, working seamlessly with the already existing OpenMP implementation. The initial Bowtie step which maps reads to Inchworm contigs has been parallelised by splitting the contigs across MPI processes. Parts of GraphFromFasta and ReadsToTranscripts which were written in OpenMP for shared memory have been changed to a hybrid implementation using MPI across distributed nodes, and OpenMP threads within a node.

We have tested the MPI-enabled hybrid version across a number of datasets, comparing the quality of the resulting transcript as well as the time taken. Repeated runs of the shared-memory version of Chrysalis show a distribution of metrics of the transcriptome, due to the stochastic nature of some of the assembly steps. The results from the MPI-enabled version also show a distribution, which overlaps the shared-memory distribution and is not significantly different.

In Section 3.2, we summarise the algorithms underlying the different components of the Trinity pipeline, along with benchmarking of Trinity to illustrate the computational requirements. Next, in Section 3.3 we give a detailed description of our MPI parallelisation scheme and its implementation, and Section 3.4 gives details of our validation method. Results are given in Section 3.5, and we conclude in Section 3.6 with an outlook on future improvements.

3.2 Existing Algorithms and Implementation

De-novo transcriptome assembly does not depend on a reference genome, instead depending on the redundancy of short reads to find sufficient overlaps between the reads. It subsequently uses these overlaps to assemble a set of transcripts corresponding to expressed genes. For a comprehensive overview of the transcript reconstruction methods for RNA-seq, please

refer to [73].

3.2.1 Trinity modules

The Trinity assembler is a heterogenous workflow comprised of modules (or software programs), which when run one after the other through a single Perl script (*Trinity.pl*), produces the reconstructed transcriptomes as the final output. In recent years, Trinity has been converted to a modular platform, using third-party tools that can be swapped in and out in future releases. The software modules exchange data through files; the files being output from one software module are then consumed by the following module. It is to be noted that Trinity also includes tools such as RSEM [34], edgeR [62] etc. that take the output of the Trinity workflow and estimate levels of gene expression, in particular for differential expression analysis. We do not include the description of those tools in this paper. For information on these tools, please refer to [21]. As mentioned earlier, Trinity's assembly pipeline consists of four consecutive modules: Jellyfish, Inchworm, Chrysalis and Butterfly. We provide a description and the function of each component below: for more details, please refer to [18].

- Jellyfish: The first step in the Trinity workflow is Jellyfish, which is a tool for fast, memory-efficient counting of k-mers (substrings of length k) in DNA [41]. Jellyfish can read FASTA and multi-FASTA files, outputting it's k-mer counts in a binary format. In the Trinity workflow, *jellyfish count* which outputs the counts of k-mers is followed by *jellyfish dump*, which converts the binary format into text format. The output of the two Jellyfish commands are thus files containing information on all k-mers extracted from the short reads with their counts. Jellyfish can output a single or multiple files depending on the available memory of the system. Jellyfish's output can be extremely voluminous - for example for the sugarbeet dataset for which we provide benchmarking results, the RNA-seq fasta file size is 15 GB, while the Jellyfish output is greater than 100 GB. Another application for k-mer counting that uses less memory than Jellyfish is DSK [61]; however this is not part of the Trinity pipeline yet. Jellyfish's output is consumed by Inchworm, the next step in the Trinity pipeline

workflow.

- Inchworm: Jellyfish's output of k-mers and k-mer counts is read by Inchworm as a first step. Since Jellyfish's output can be voluminous, the reading of the k-mers into Inchworm can also take a substantial amount of time. Inchworm constructs a hash table object consisting of pairs or duals - the duals are comprised of the k-mers along with the read abundance of the kmer. Constructing the hash table object from the k-mer file using multiple OpenMP threads can be both time and memory intensive - since Inchworm keeps this entire hash table object in memory, Inchworm's memory footprint can be extremely high. This hash table object is subsequently sorted in order of decreasing k-mer abundance. Inchworm examines each unique k-mer starting from the most abundant, and generates Inchworm contigs using a greedy extension based on (k-1)-mer overlaps. These contigs are subsequently written to disk. Inchworm thus goes through the following steps:

  - Constructs a k-mer dictionary from all sequence reads removing likely error-containing k-mers, and sorts them in decreasing order of abundance.
  - Selects the most frequent k-mer in the dictionary to seed a contig assembly.
  - Extends the seed in each direction by finding the highest occuring k-mer with a k-1 overlap as shown in Figure 11.
  - Extends the sequence in either direction until it is not extended further, reporting the linear contig.
  - Repeats these steps with the next abundant k-mer until the entire dictionary is exhausted.

  In summary, Inchworm reads in the massive k-mer file written by Jellyfish, does a greedy extension of the kmers in decreasing order of abundance (consisting of the steps above), and then writes a comparatively much smaller file of contigs.

- Chrysalis: Chrysalis clusters minimally overlapping contigs obtained from Inchworm into separate sets of connected components, followed by construction of de Bruijn

graphs for each component. Chrysalis itself is composed of two separate modules - *GraphFromFasta* and *ReadsToTranscripts* - which specifically do the following:

  - *GraphFromFasta* clusters related Inchworm contigs into so-called components. It does this by welding pairs of contigs together if read support exists, and subsequently clustering Inchworm contigs using these welds and building de Bruijn graphs for each component. Both of these steps are already parallelized with OpenMP threads, but since each possible pair of Inchworm contigs has to be compared, this process can still be extremely compute-intensive.
  - *ReadsToTranscripts* assigns each read to the component with which it shares the largest number of k-mers, as well as determining the regions within each read that contribute k-mers to the component.

  Apart from the components mentioned above, Trinity also uses *Bowtie* (a third-party tool) to align input reads to Inchworm contigs.

- Butterfly: Butterfly reconstructs feasible full-length linear transcripts by reconciling the individual de Bruijn graphs generated by Chrysalis with the original reads and paired end data. Each Chrysalis component or graph can produce several linear transcripts, which in most cases will correspond to alternative splicing of the gene product.

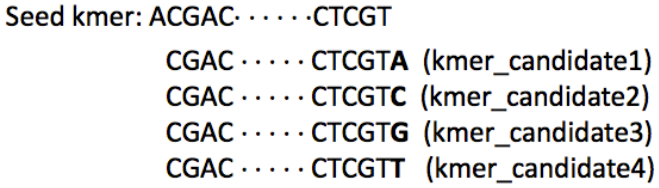

Figure 11: Seed extension by k-mer with (k-1) overlaps.

3.2.2 Benchmarking of Original Trinity

To understand the basic characteristics of Trinity performance, we measured memory usage and runtime of each step in Trinity using the Collectl tool [10] distributed with Trinity. To perform this run, Trinity was compiled using GNU compilers and the performance evaluated

using a sugarbeet RNA-seq dataset kindly provided by Rothamsted Research, UK. The dataset is 15 GB in size on disk and contains 129.8 M reads, with two subsets of 9 GB (79.2 M single end and left reads) and 6 GB (50.6 M right reads). Our sugarbeet dataset is larger than a typical test dataset in order to illustrate the computational challenges. Nevertheless, it is only representative of a routine RNA-Seq experiment, and much larger datasets are now being generated by sequencing facilities. Since Trinity already came with support for multiple OpenMP threads, this initial run was done using 16 threads on a single iDataPlex node at the Hartree Centre, UK. A single iDataPlex node at the Hartree Centre comprises 2x 8 core 2.6 Ghz Intel SandyBridge processors, with 256 GB of memory. Figure 12 shows the results of this original Trinity run, showing the RAM usage on the Y-axis with the runtime (in hours) along the X-axis.

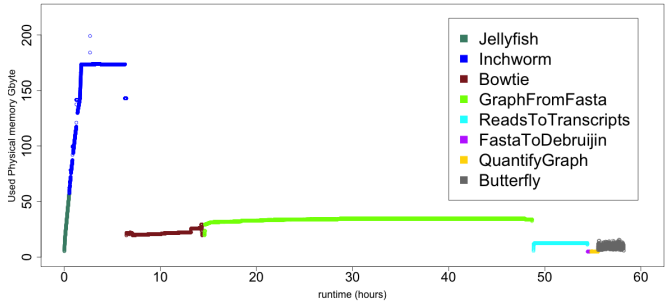

Figure 12: Measurement of RAM usage (Y-axis) and the runtime (X-axis) of Trinity workflow run using single node of 16 cores and 256 GB of memory for the sugarbeet dataset.

As can be seen from Figure 12, it is clear that even for sugarbeet dataset, the runtime of the entire Trinity pipeline is close to 60 hours. Chrysalis is seen as the most time-intensive phase of the Trinity pipeline. The Chrysalis phase itself is composed of several sub-steps: Bowtie, GraphFromFasta, ReadsToTranscripts, FastaToDebruijn and QuantityGraph. Most of the runtime in Chrysalis is in three steps: Bowtie, GraphFromFasta and ReadsToTranscript, which appear in the same order in the Chrysalis step. For this reason, we decided to focus our parallelization efforts mainly on these three components. GraphFromFasta and ReadsToTranscripts are already parallelized for a shared-memory architecture with OpenMP threads. Thus, our efforts were focused on a distributed memory

implementation of these components, that works with the existing OpenMP implementation. For Bowtie, we leverage PyFasta [57], that can be used to split the target sequences amongst the MPI processes, thus not requiring any source code changes. In the next Section 3.3, we include details on our parallelization, with the subsequent computational speedups reported in Section 3.5.

3.3 Parallelization of Trinity workflow

As can be seen from the previous description of Trinity modules and benchmarking results, Trinity software is meant for usage only on a shared-memory machine. To improve the performance of the Trinity workflow, we focused on the most compute-intensive parts of Trinity which includes *Bowtie*, *GraphFromFasta* and *ReadsToTranscripts*.

We also had to be careful that our additional MPI code works well with the existing OpenMP code that is already being used for most of the time-intensive loops in Chrysalis; thus the OpenMP sections had to be changed to a hybrid model using MPI across multiple nodes, and OpenMP within a node. In the subsections below, we describe the changes made for the MPI implementation of Bowtie, GraphFromFasta and ReadsToTranscripts. We detail our parallel implementations below in the order that they are run in the Chrysalis workflow.

3.3.1 MPI implementation of Bowtie

The main objective of Chrysalis is building Inchworm bundles where each bundle is a cluster of Inchworm contigs. The Inchworm contigs in the same bundle are used for full reconstruction of transcripts. To build Inchworm bundles, Chrysalis first aligns input reads to Inchworm contigs using Bowtie. Based on the output from Bowtie alignment, the subsequent step searches pairs of Inchworm contigs of which both ends are to be combined for the construction of scaffold, provided that some of input reads are aligned onto single end of each contigs. This output is later combined with "welding" pairs of Inchworm contigs from GraphFromFasta for full construction of Inchworm bundles. More details of "welding" pairs of Inchworm contigs are described in the following section.

Bowtie already has an option for using multiple threads simultaneously on a single node to achieve a faster alignment speed. However, with millions of input reads, it can require several hours of runtime as shown in Figure 3. To speed up the alignment process, we ran Bowtie on multiple nodes by splitting the target sequences of Bowtie, i.e. the Fasta file of Inchworm contigs. The Fasta file was partitioned using the PyFasta python module, which evenly splits the target sequences amongst the rank nodes for parallel alignment processing. Each node then produces an alignment output file in SAM format, and the files from all nodes are merged into a single file at the end of the job. Our approach is different from [4], which did not use MPI, but looked at different partitioning of reads and genomes over nodes. Our partitioning of the Inchworm contigs over nodes is a special case of their more general study.

3.3.2 MPI implementation of GraphFromFasta

GraphFromFasta contains two compute-intensive loops. The first loop goes through each Inchworm contig; first, it finds all possible k-mers from the current contig, and subsequently it harvests "welding" subsequences which match sub-regions of other contigs. The size of the welding subsequence is $2k$ consisting of the seed k-mer and left- and right-flanking $\frac{k}{2}$-mers. That is, the first loop decides if common subsequence exists to "weld" two Inchworm contigs into the same Inchworm bundle at the end of GraphFromFasta. The loop is already multi-threaded with OpenMP threads; since the work done per Inchworm contig is not uniform (depending on the contig, either it is welded with other nchworm contigs or not), the OpenMP scheduling policy is *dynamic*. Each thread gets multiple Inchworm contigs, working on them until they run out, at which point they again get multiple contigs. The "chunksize" or the number of Inchworm contigs processed by each OpenMP thread is proportional to the number of Inchworm contigs divided by the number of threads.

Our focus was to change this loop into a hybrid loop, with additional speedup coming from the use of multiple nodes, each running multiple OpenMP threads. In the beginning, we pre-allocated chunks of Inchworm contigs to each MPI process. However, this did not give us a good speedup, especially when using the multiple threads. Our current implementation

uses a "chunked round robin" strategy with each MPI process getting a chunk, distributing to its multiple threads, and then working on the next chunk. Mathematically, in the outer loop, chunk i consisting of n Inchworm contigs is allocated to MPI rank p if $i(modulo)p = 0$. The chunk consisting of n contigs is subsequently divided amongst the OpenMP threads in an inner loop. A contig's data is thus accessed by the sum of the index of the chunk with the index of the contig within the chunk. The OpenMP scheduling strategy is kept as dynamic as in the original loop. We had to be careful with such a strategy however, as there might be the case that some MPI processes might still try to get a full chunk, even though the number of Inchworm contigs left is less than the chunk size. Thus, the end index of the inner thread loop might have to be changed depending on how many Inchworm contigs are left for the MPI process. Figure 13 shows our "chunked round robin" distribution strategy.

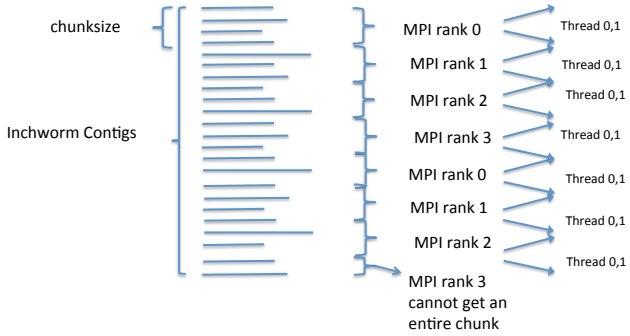

Figure 13: Chunked round robin strategy for hybrid MPI + OpenMP code with 4 MPI processes and 2 OpenMP threads as an example.

Once all the MPI processes are done, they have a vector of the "welding" subsequences, which have to be pooled together on each rank from every rank before the second loop. As a first step, the vector of the subsequences are packed into a single sequence for MPI communication. Consequently, each MPI process then exchanges the size of this packed sequence to every other rank for subsequent communication using *MPI_Allgatherv* which pools together the sequences on every rank. At the end of the first loop, every rank, thus has a pool of sequences combined from every other rank. This pool of sequences is subsequently used for the second loop.

The second compute-intensive loop finds pairs of Inchworm contigs sharing any "welding" subsequence harvested from the first loop. Our "chunked round-robin" strategy for the

distribution of Inchworm contigs is also used in this loop. The output of the second loop is the list of indices for pairing Inchworm contigs for "welding", which is pooled on every rank in a similar way as in the first loop. First, the integer values for pairing indices are packed into single integer array for MPI communication. Subsequently, each MPI process exchanges the size of this integer array to other ranks for communication which is then used for pooling the pairing indices together on every rank. Since only integer arrays are exchanged, the second loop uses substantially less communication compared to the first loop where vectors of strings are exchanged between rank nodes.

3.3.3 MPI implementation of ReadsToTranscripts

ReadsToTranscripts assigns each input read to the Inchworm bundle against which the largest number of its constituent k-mers align. Since ReadsToTranscripts works with the input reads file which can be extremely large, ReadsToTranscripts does not try to load the entire input reads file into memory, but instead relies on a streaming reads model. This is opposite to GraphFromFasta; since GraphFromFasta only works with the Inchworm contig file which can be much smaller, it reads the entire file into memory. ReadsToTranscripts uploads chunk of input reads from the file into memory depending on a command-line parameter *max_mem_reads* which decides the number of input reads uploaded into memory at a time. These reads are inserted into a vector of strings, which is then used in the compute-intensive part of the loop. This part links each input read to the Inchworm bundle for which the largest number of its possible k-mers are aligned. This compute intensive loop is also OpenMP enabled with the set of uploaded input reads distributed over the OpenMP threads.

For a hybrid implementation, it was obvious that we needed to parallelize the uploading of the short reads across the multiple MPI processes. One of our strategies was to let only a master node or rank read the sequences and distribute to the other "slave" nodes. However, this strategy involves relatively heavy communications between master and slave nodes which leads to a bottleneck particularly as the number of slave nodes increases. Our second updated implementation allowed every rank to read the *max_mem_reads* by counting

the number of chunks of the *max_mem_reads* uploaded into memory by a MPI process. If this count value is not a multiple of the rank, then the MPI process simple discards the uploaded input reads, and then reads another chunk of the input reads. This process continues until the count value is a multiple of the process's rank, at which point the reads are distributed amongst the OpenMP threads of this MPI process. This approach does make every process read redundant data (each process in fact reads the entire file), but excludes the necessity of MPI communication.

At the end of ReadsToTranscripts, a file with information on the reads aligned to the Inchworm contigs is written by each process. There is a final command at the end by the master node which combines the multiple files into a single file with a simple *cat* command. We have found the overhead of this concatenation step to be fairly low; another option is merging the data at the root process from all the processes and only let the root process write the final output.

Our current software methodology works as follows: *Trinity.pl* which is the Perl script that calls all the Trinity components has been extended with an argument for the number of processes (*nprocs*). This argument is then used in the command-line for the Chrysalis executable, which calls GraphFromFasta and ReadsToTranscripts separately from within it's source code. If the source code is compiled with MPI support primitive enabled, the command line for GraphFromFasta and ReadsToTranscripts is prepended with a suitable MPI runtime mechanism (such as *mpirun -np nprocs*) that allows both of these software modules to be run with multiple processes. In Section 3.4, we show the validation methodology of our parallel implementation. In Section 3.5, we show the performance results obtained by the MPI enabled GraphFromFasta and ReadsToTranscripts, as well as the distributed version of Bowtie.

3.4 Validation of Parallel Trinity

To show that the hybrid parallelized Trinity produces equivalent results in the reconstruction of transcripts to the original version of Trinity, we performed two sets of validation

tests. These tests indicate that there is no significant difference in the output between both versions of Trinity. It should be noted that Trinity produces slightly indeterministic [21] output, in which the outputs from multiple runs using the same input data set can be slightly different, and therefore we do not expect identical results between both versions of the code.

The first test is an all-to-all sequence alignment approach, in which all reconstructed transcripts from the hybrid parallelized Trinity were aligned to those from the original Trinity using the Smith-Waterman algorithm, as implemented in the FASTA program [50]. Due to the indeterministic nature of Trinity, multiple results from ten repeated runs for each version of Trinity (OpenMP-only and MPI+OpenMP) were obtained. In addition to aligning transcripts between the different versions of Trinity, we also aligned transcripts from the different runs of the original Trinity, in order to understand the expected level of variation in the output. The alignment results using a whitefly data set downloaded from the internet [14] comprised of a total of more than 420,000 reads with left and right reads of approximately 210,000 each are shown in Figure 14. They show no significant difference between the two versions of the code according to a two sample t-test, thus indicating that the output from the hybrid-parallelized Trinity has equal quality to the one from original version of Trinity.

The second test involves measuring the number of reconstructed transcripts identified as known transcripts. The number was simply measured by aligning the reconstructed transcripts obtained from runs of Trinity against a set of reference transcripts, and this number was compared between the two versions of Trinity. The reference transcripts are comprehensive and well-annotated sets of transcript sequences, obtained from the Trinity FTP site for the Schizophrenia and Drosophila datasets. The Schizophrenia dataset consists of 9.2 million left reads and 6.15 million right reads, for a total of 15.35 million reads with a size of about 8 GB on disk. The Drosophila dataset consists of 50 million left and right reads, with a total size of about 10 GB on disk. Four related numbers were counted in this test:

- The number of genes of which at least one reconstructed isoform was aligned in full

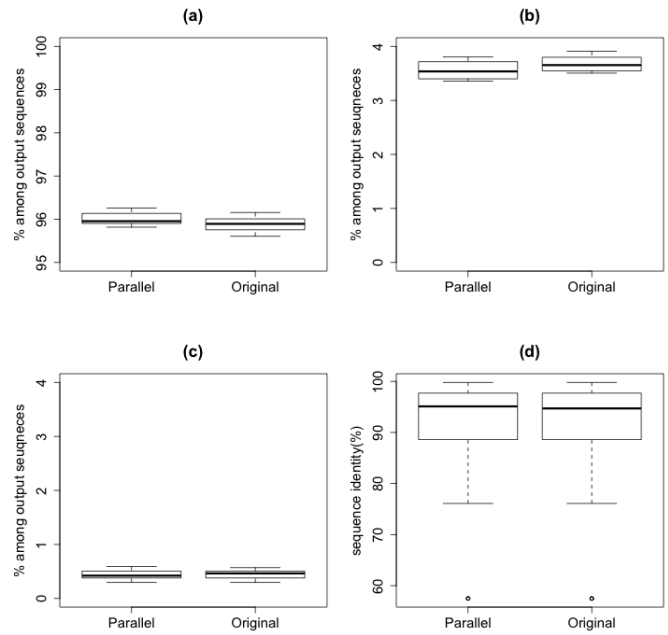

Figure 14: Alignment of the reconstructed transcripts from parallelized Trinity to the ones from original Trinity using Smith-Waterman algorithm in FASTA program using whitefly dataset. The results are categorized into three groups; (a) 100% identical match for full length, (b) less than 100% identical match for full length and (c) less than 100% identical match for partial length. The distribution of identities/similarities of aligned sequence pairs in (c) is described in (d). "Parallel" represents the sequence alignment of two sets of reconstructed transcripts from parallelized Trinity and original Trinity, respectively. "Original" represents the alignment of two sets of reconstructed transcripts from original and original Trinity.

length onto one of the reference transcripts (see graphs (a) and (c) of Figure 15).

- The number of reconstructed isoforms aligned in full length onto one of the reference transcripts (see graphs (b) and (d) of Figure 15).

- The number of genes of which at least one reconstructed isoform corresponds to a fusion of multiple full-length reference transcripts (see graphs (a) and (c) of Figure 16).

- The number of reconstructed isoforms which correspond to a fusion of multiple full-length reference transcripts (see graphs (b) and (d) of Figure 16).

The "fused" transcripts considered in the last two numbers are single reconstructed transcripts including multiple full-length transcripts from the reference set. These transcripts are reconstructed as end-to-end fusions in some cases due to overlapping UTRs or other

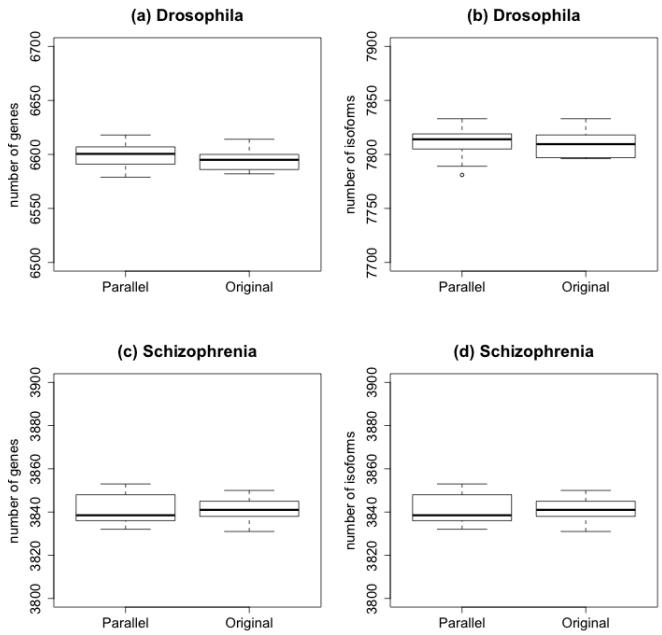

Figure 15: Alignment of reconstructed transcripts from both versions of Trinity to the reference transcripts; number of fully reconstructed genes/isoforms in full-length for Schizophrenia (a, c) and Drosophila (b, d) datasets among the reference transcripts. Note the number of reconstructed genes in full-length represents the case that at least one isoform is reconstructed in full-length.

factors. These are likely false-positive reconstructed transcripts; however, these are still counted separately as being reconstructed transcripts due to their full length. Comparing these four numbers indicates that there is no significant difference in the outputs from both versions of Trinity.

3.5 Results

In this Section, we report the results obtained by running the distributed memory versions of GraphFromFasta, ReadsToTranscripts and Bowtie. The MPI enabled source code was compiled with OpenMPI 1.6 which is an open-source MPI implementation, using the GNU compiler version 4.4.6. Our test hardware is an iDataPlex cluster, known as "Blue Wonder", comprising 512 nodes each with 2x 8 core 2.6 GHz Intel SandyBridge processors making 8,192 cores in total. Out of these 512 nodes, 256 nodes have 128 GB of memory which are the nodes we used for the MPI benchmarking.

The input dataset we used for the benchmarking is the sugarbeet dataset, which is

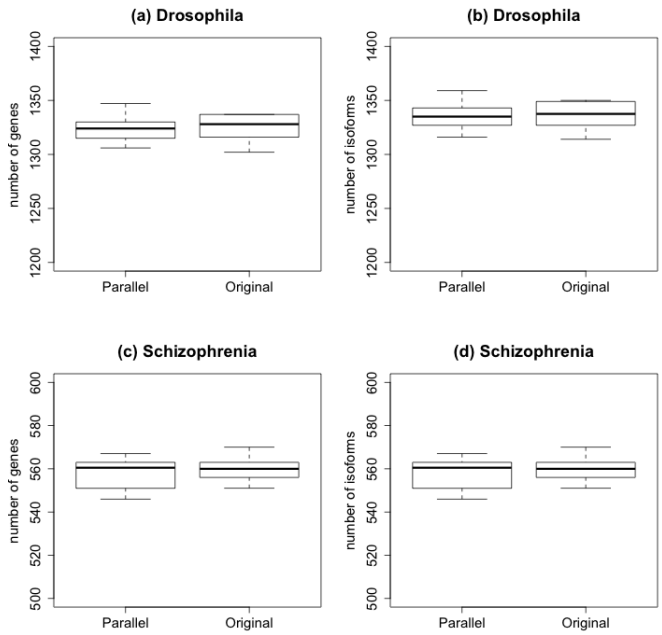

Figure 16: Alignment of reconstructed transcripts from both versions of Trinity to the reference transcripts; number of reconstructed genes/isoforms in full-length as "fused" transcript for Schizophrenia (a, c) and Drosophila (b, d) datasets. Note "fused" transcript is defined as single reconstructed transcript including transcripts from multiple genes/isoforms.

15 GB in size on disk and contains 129.8 M reads, with two subsets of 9 GB (79.2 M single end and left reads) and 6 GB (50.6 M right reads). The same dataset was used in the original benchmarking of Trinity (see Figure 12).

3.5.1 Hybrid (MPI+OpenMP) GraphFromFasta

Figure 17 shows the results of the MPI enabled hybrid GraphFromFasta code using the sugarbeet dataset as the input. The number of processes was varied from 16 to 192; each node runs a single MPI process, with 16 OpenMP threads. We started the runs with 16 nodes as the runtimes below 16 processes exceeded the maximum queue time of the parallel jobs. This graph shows the time taken separately in loops one and two, both of which were converted to a hybrid implementation, along with the total time taken in GraphFromFasta. Along with the loops one and two, GraphFromFasta also consumes time in other tasks setting up the k-mers before the second loop and generation of the final output after the second loop. As explained in Section 3.3, loop one decides if a common subsequence exists to "weld" two Inchworm contigs into the same Inchworm bundle, while the second loop

finds pairs of Inchworm contigs sharing any "welding" subsequence harvested from the first loop. This figure shows the lowest and the highest time taken in the loops, amongst all the MPI ranks, as a measure of load imbalance.

For all performance analysis, we consider the representative time as the processes with the highest times. For loop one, at 128 and 192 nodes, using data from the nodes with the highest time, we get a speedup of 8.31 and 11.93 compared to time of the loop from 16 nodes. For loop two, the speedups are 7.62 and 5.64 respectively using the loop timings at 16 nodes. At 192 nodes, the speedup of loop two is primarily lower due to load imbalance with the highest time of a process more than three times the process with the lowest time. For loop one as well, the highest MPI rank time is 50% higher than the lowest MPI rank time for the same number of nodes. Some of this load imbalance is due to the nature of the problem: there is a very wide variation in the lengths of reconstructed transcripts with some lengths being in tens of thousands, while others only a few hundred characters. This leads to an imbalance in the amount of work each node has to carry out. Currently, we have a static partitioning strategy amongst the nodes; in the future, we might experiment with a dynamic partitioning strategy to reduce this load imbalance.

For the entire GraphFromFasta, the baseline performance is the performance measured with the OpenMP only version run with 16 threads on one node (122610 seconds). The time taken by 16 nodes, each running 16 threads, is 27133 seconds, while with 192 nodes, each employing 16 threads, the total runtime decreases to 5930 seconds. These runtimes correspond to speedups of 4.5 and 20.7 respectively for the GraphFromFasta overall. This low speedup is primarily due to the share of the non-MPI regions accounting for a increased percentage of the total GraphFromFasta time with increasing nodes, for example at 16 nodes, the time taken by both the loops comprises 92.44% of the total time of GraphFromFasta, which falls to 57.4% at 192 nodes. Figure 18 shows the breakup of the GraphFromFasta times into separate timings of loop 1, 2 and the non-parallel regions. As can be seen, non-parallel regions account for an increasing percentage of the total time of GraphFromFasta; at 128 processes, the total percentage of time taken by the non-parallel regions accounts for 63.3% of the total time of GraphFromFasta. At 192 nodes, the load imbalance especially in

loop 2 leads to the share of the non-parallel regions decreasing. Our future work will also involve parallelizing other parts of GraphFromFasta, as well as reducing the load imbalance which will further help speed up the overall time of execution.

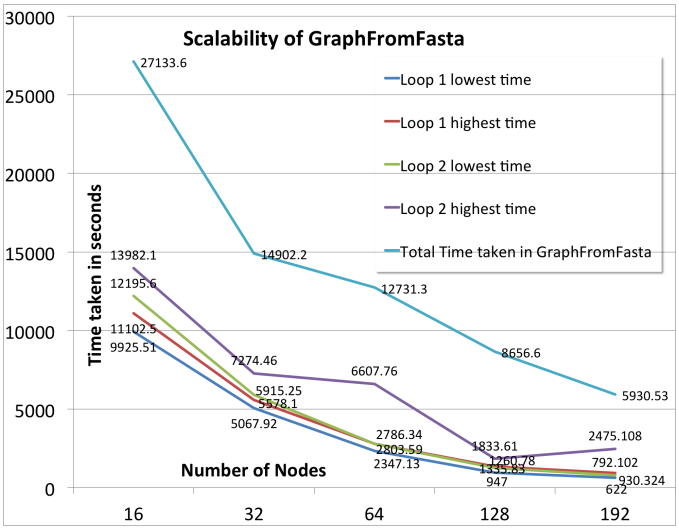

Figure 17: Results of parallel (MPI+OpenMP) GraphFromFasta implementation showing the time taken in the loops and the total time taken in GraphFromFasta with increasing number of nodes.

3.5.2 Hybrid (MPI+OpenMP) ReadsToTranscripts

Figure 19 shows the results of the hybrid ReadsToTranscripts, the second part of the Chrysalis module. We again use the sugarbeet dataset as the input. We also continue the mode of execution of running 16 threads per node, with a single MPI rank, which has been shown to give the best performance. We show the time taken in the main loop which was MPI-enabled, together with the total time taken in ReadsToTranscripts. Besides the MPI-enabled loop, ReadsToTranscripts also spends time in assigning k-mers to Inchworm bundles as well as concatenation of the separate files from every process. The assignment of k-mers to Inchworm bundles is OpenMP-enabled, and we have not converted this to a hybrid implementation yet.

At 32 nodes, the total runtime of ReadsToTranscripts takes less than 20 minutes: thus, this step of Chrysalis does not represent as significant a computational overhead as GraphFromFasta and Bowtie. The scalability of the MPI loop is almost linear, from about 3123 seconds on 4 nodes to less than 373 seconds on 32 nodes, representing a speedup of 8.37.

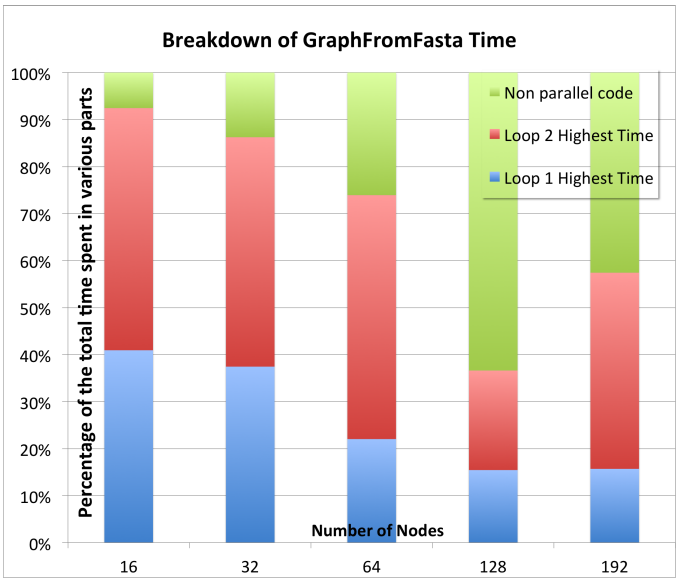

Figure 18: Breakdown of GraphFromFasta times showing the times taken in loop 1, 2 and non-parallel regions. All times are normalized to 100%.

At 32 nodes, the percentage of time spent in the MPI loop represents less than 20% of the total time spent in ReadsToTranscripts with the remaining time primarily taken in the OpenMP-enabled assignment of k-mers to Inchworm bundles. On a single node, the ReadsToTranscripts, using 16 threads took a total runtime of 20190 seconds. At 32 nodes, we thus achieve a overall speedup of 19.75 for the entire ReadsToTranscripts execution.

A very small percentage of the time is taken in the concatenation of the files from the multiple processes: this time stays constant (below 15 seconds) atleast up to 32 processes. As in Figure 17, we have also shown the processes with the highest and lowest times (373 and 310 seconds) spent in the loop: thus, compared to GraphFromFasta, the load imbalance in ReadsToTranscripts is much lower.

3.5.3 MPI Implementation of Bowtie

Figure 20 shows the results of a scaling experiment for the parallelized Bowtie again using the sugarbeet input dataset. This run was also completed using 16 threads, with one MPI rank per node. Since Bowtie with multiple nodes requires splitting the Fasta file of Inchworm contigs, we include runtimes for Fasta file splitting using PyFasta and the actual runtime for MPI-Bowtie, as well as the total Bowtie runtime. The figure shows that the

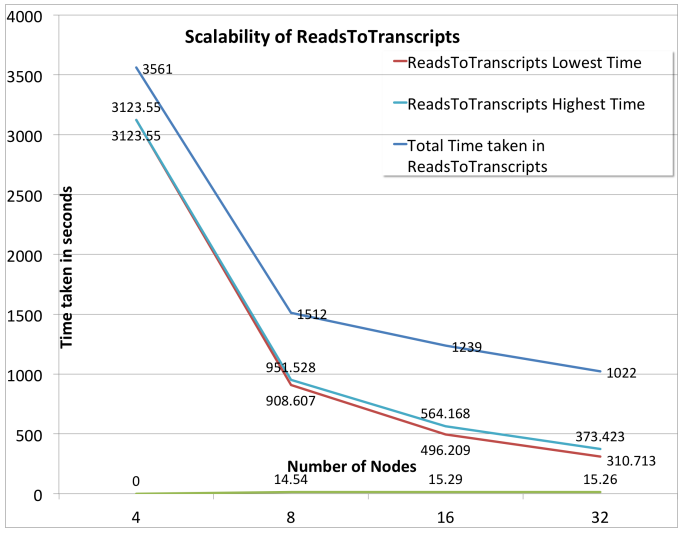

Figure 19: Results of parallel (MPI+OpenMP) ReadsToTranscripts implementation showing the time taken in the main loop and the total time taken in ReadsToTranscripts with increasing number of nodes.

splitting of the Fasta file using PyFasta took more runtime than the subsequent Bowtie step, partially due to the fact that PyFasta is a single thread process. We consider this step as a possible overhead to be worked on for better performance. In summary, we got a speedup of a factor of three when Bowtie was implemented in parallel using 128 nodes compared to single node implementation which took slightly more than 8 hours.

Overall, the Trinity workflow execution with the parallel Bowtie, GraphFromFasta and ReadsToTranscripts is shown in Figure 21. We again used the Collectl tool to collect statistics from the run, using 16 nodes, each running a single MPI process with 16 threads. This figure, compared to Figure 12, shows the substantially lower time taken in Chrysalis workflow using the sugarbeet dataset as the input.

3.6 Summary

In this chapter, we have focused on a distributed-memory implementation of Chrysalis components GraphFromFasta and ReadsToTranscripts, with the goal of producing an MPI

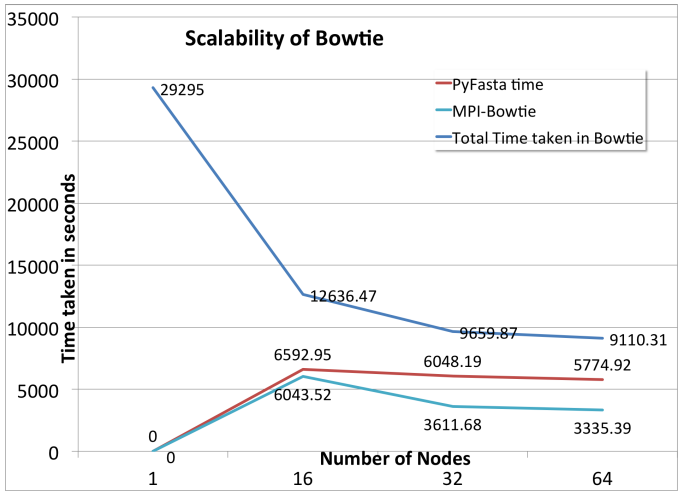

Figure 20: Results of parallel Bowtie implementation showing the time taken in Bowtie and time taken by PyFasta to partition the Fasta file.

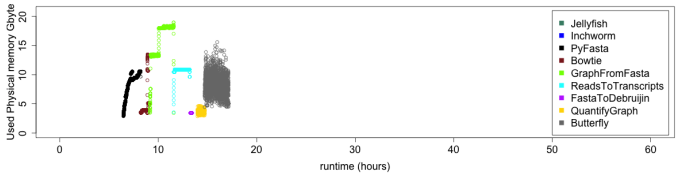

Figure 21: Parallel Trinity run using 16 nodes, each with 16 cores and 128 GB of memory. Running instances of Inchworm/Jellyfish are not recorded for MPI-parallelized Trinity

implementation working seamlessly with the OpenMP threads that is already part of both the software modules. Overall for the sugarbeet dataset, we have reduced runtimes of GraphFromFasta and ReadsToTranscripts by over a factor of 20. The speedup comes essentially from the ability to use multi-node architectures, as widely available in traditional clusters, rather than relying on a single high-performance workstation. This fundamental change will allow Trinity to tackle the ever-increasing datasets that are being collected.

CHAPTER IV

K-MER CLUSTERING ALGORITHM USING A MAPREDUCE FRAMEWORK

4.1 Background

Quantifying the expression of genes under different conditions is fundamental to understanding the behaviour and response of organisms to internal and external stimuli. With the arrival of Next Generation massively parallel sequencing technologies, the ability to monitor gene expression has been transformed [11] [67]. Direct sequencing of mRNA from expressed genes (RNA-Seq) is now feasible, and has several advantages over microarray technology [49]. Most notably, it removes the need to have a priori knowledge of the transcribed regions, so that novel genes can be identified, or novel variants of known genes. This has led to a rapid increase in the number of studies looking at gene expression in non-model organisms. RNA-Seq is also increasingly used to study non-coding RNAs, such as microRNAs [20], lincRNAs [75], and circRNAs [45] which play various regulatory roles.

Nevertheless, it is widely recognised that the improvement in sequencing technology has shifted the bottleneck to down-stream data analysis. In the case of RNA-Seq, sequencing can be complicated by the presence of contaminant RNA, paralogous genes, and especially for higher organisms the prevalence of alternative splicing [59] [47]. Paired-end sequencing and strand-specific sequencing can help to resolve sequencing ambiguities, but must be included explicitly in the data analysis. Finally, and as we address in this study, the sheer size of datasets can cause practical problems in sequence assembly. In particular, the computational complexity due to the typical size of RNA-Seq datasets limits the ability to try multiple methods or multiple parameter choices, in order to optimise the quality of the results obtained. Initial approaches to the high throughput analysis of transcriptome sequence data were based on the alignment of RNA-Seq reads to reference genomes [47] [15]. Such approaches are limited by the availability of suitable reference genomes, and by the structural

alterations that can be detected, particularly when input reads are relatively short. Subsequently, de novo genome assemblers were adapted to the analysis of transcriptome data in the absence of a reference, by postprocessing draft contigs to identify transcripts. Examples of transcriptome assemblers based on genome assemblers include Oases [68] and postprocess [74] based on Velvet [83], TransABySS [1] based on ABySS [71], and SOAPdenovo-Trans [79] based on SOAPdenovo [36]. In contrast, the Trinity [18] pipeline which we consider below was developed specifically for de novo transcriptome assembly. More recent examples hybridizing previous de novo assembly algorithms include Bridger [8] based on Trinity [18] and SOAPdenovo-Trans [79], BinPacker [37] based on Bridger [8] and bin-packing strategy [42], and DRAP [6] based on Trinity [18] and Oases [68].

Most de novo transcriptome assembly methods are based on de Bruijn graphs of k-mers, where a k-mer is a sub-sequence of an input read with k base calls. For a chosen value of k, the assembler creates a k-mer graph, where the set of nodes correspond to all unique k-mers present in the input reads, and the edges represent "suffix-to-prefix" overlaps between k-mers. Most de novo transcriptome assembly algorithms store all unique k-mers from the input reads in shared memory, in order to facilitate edge detection and graph construction, and this can lead to extremely large RAM usage [84]. For example Velvet, as used by Oases, starts by creating two large hashmap tables in memory storing the information for all k-mers. TransABySS/ABySS is the only parallel algorithm for de novo transcriptome assembly, which starts by distributing k-mers onto multiple compute nodes with a simple hash function. The Trinity pipeline consists of three independent software modules; Inchworm, Chrysalis and Butterfly. Inchworm initially creates a large hashmap table to store all unique k-mers from the input RNA-seq reads, and then it selects k-mers from the hashmap to construct linear contigs using a greedy k-mer extension approach. Chrysalis clusters Inchworm contigs into sets of connected components that are linked by pair-end reads, and creates de Bruijn graphs for each set. Butterfly then reconstructs the full-length transcripts based on the de Bruijn graphs from Chrysalis, taking into account possible alternative splicing. In our previous study [63], we identified the Chrysalis module as the main bottleneck in terms of runtime, and alleviated this bottleneck by parallelising the

processing over multiple compute nodes using MPI. We also confirmed that the Inchworm module of Trinity requires relatively high physical memory usage.

The memory requirements of these packages increase for larger and more complex transcriptomes, which generate larger numbers of k-mers and hence larger graphs, and can exceed the computational resources available. One strategy that is commonly used is to normalize the read data [5]. Redundant reads are removed from regions with high sequencing coverage, while reads are retained in regions of low coverage. In this way, up to 90% of input reads can be removed, which in turn leads to the elimination of a large fraction of erroneous k-mers associated with these reads [5]. While this is believed to work well, it introduces an additional processing step, which can in itself require large memory.

The fundamental task of de novo transcriptome assembly (in contrast to genome assembly) is to separate the full sequence data into many disjoint sets. Each set corresponds to a collection of gene variants sharing k-mers due to alternative splicing or gene duplication. In other words, a transcriptome can be represented as multiple distinct de Bruijn graphs, each of which contains several paths corresponding to alternative gene products. Intuitively, de novo transcriptome assembly could be performed for every connected sub-graph separately. In the case of genome-guided transcriptome assembly, generation of sub-graphs is directed by the reference genome. In the absence of such a method for de novo assembly, however, most assemblers [67] [79] [18] work with all unique k-mers obtained from the input reads, resulting in the requirement for a large amount of available memory. In this work, we present a reference-free method for generating connected sub-graphs from datasets of RNA-Seq reads. We employ the MapReduce formulation [12] for distributing the analysis of large datasets over many compute nodes. The MapReduce approach was popularized by Google for handling massively distributed queries, but has since been applied in a wide range of domains, including genome analysis [44] [46] [80]. A typical MapReduce implementation is based on map() and reduce() operations that work on a local subset of the data, but the power of the approach comes from an intermediate step called shuffle() or collate() which is responsible for re-distributing the data across the compute nodes. In the context of transcriptome assembly, the MapReduce approach distributes the sequence data

over the available nodes, thus reducing the per-node memory requirement. The iterative application of map(), collate() and reduce() steps leads to clustering of the k-mers, such that the desired sub-graphs are each physically located on a single compute node.

While distributing the sequence data across nodes of a compute cluster should lead to faster runtimes and reduced per-node memory requirements, this must be balanced against the cost of inter-node communication and transfer of data. We make use of an established MapReduce software library [54] that handles communication via the Message Passing Interface (MPI) protocol. Using this library, we have developed software that can cluster k-mers, and then launch multiple Inchworm jobs for the resulting sub-graphs. The procedure can be linked with the rest of the Trinity pipeline, for selected components of which we have also developed an MPI-based parallelisation [63], so that the entire assembly workflow can be run on a commodity cluster. Use of the MapReduce-MPI software library [54] means that specialised MapReduce installations such as Hadoop are not required. The only requirement is an MPI installation which, while requiring some setup and management, is well-established and omnipresent on high performance computing platforms.

4.2 Methods

4.2.1 MapReduce-MPI library

The MapReduce [12] programming paradigm consists of two core operations, namely a "map" operation followed by "reduce" operation. These are highly parallel operations working on distributed data, which wrap around an intermediate data-shuffling operation that requires inter-processor communication. The basic data structures for MapReduce operations are key/value (KV) pairs, and key/multivalue (KMV) pairs that consist of a unique key and a set of associated values. There are many implementations of the MapReduce idea, see for examples [77] [58]. In the MapReduce-MPI library [54], which we utilise here, KV and KMV pairs are stored within MapReduce objects, and user defined algorithms consist of operations on these objects.

A typical algorithm using the MapReduce-MPI library is built upon three basic functions operating on MapReduce objects, namely map(), collate() and reduce(). In map(), KV pairs

are generated by reading data from files or processing existing KV pairs to create new ones. The collate() operation extracts unique keys and maps all the values associated with these keys to create KMV pairs. The reduce() operation processes KMV pairs to produce new KV pairs as input to the following steps of the algorithm. In a parallel environment, the map() and reduce() operations work on local data, while the collate() operation builds KMV pairs using values stored on all processors. Since KV pairs with the same key could be located on many different processors, there is a choice about where to store the resulting KMV pair. In the MapReduce-MPI library, each KMV pair is distributed onto a processor by hashing its key into a 32-bit value whose remainder modulo the number of processors is the owning processor rank.

The MapReduce-MPI library allows user-defined functions to be invoked for map() or reduce() operations, while the collate() operation and the general housekeeping of MapReduce objects are handled automatically. The map() and reduce() operations are called via pointers to functions supplied by the application program. Each user-defined function is invoked multiple times as a callback for each KV or KMV pair that is processed.

Out-of-core processing is an important feature of the MapReduce-MPI library, and is initiated when KV or KMV pairs owned by a processor do not fit in the physical memory. When this happens, each processor writes one or more temporary files to disk and reads the data back in when required. Specifically, a pagesize is defined by the user, which is the maximum size of MapReduce objects that can be held in memory and used in MapReduce operations. This allows the MapReduce-library to handle data objects larger than the available memory, at the expense of additional I/O to disk, and we give examples later.

4.2.2 Finding Connected Components

A connected component of an undirected graph is a sub-graph where any two nodes are connected by a path of edges. A transcriptome can be represented as a k-mer graph with multiple connected components, where ideally the number of sub-graphs equals the number of genes. The identification of connected components can be done using a depth-first search [24]. Starting from a seed node, the procedure searches for the entire connected component

by repeatedly looping through neighbour nodes, and creates new paths between nodes as extensions of pre-existing paths.

An algorithm implementing the above search in a MapReduce framework starts with the assignment of unique "zone" IDs to each graph node stored in a MapReduce object [55]. In each iteration, the size of a zone may increase by one layer of its neighbours. As zone IDs between two nodes conflict by sharing edge, a winner is chosen and the losing nodes are then merged into the winning zone. When the final iteration is reached, all nodes in a connected component will have been assigned to the same zone, and the MapReduce object will contain the zone assignments for all fully connected components. More details of the algorithm and its implementation in the MapReduce-MPI library are given in [55]. For the current application, we need to define the nodes and edges of the full (disconnected) graph to be analysed, which we do in the next subsection.

4.2.3 MapReduce-Inchworm

We have implemented a multi-step procedure for clustering k-mers as the initial stages of transcriptome assembly in Trinity [18] as shown in Figure 22.

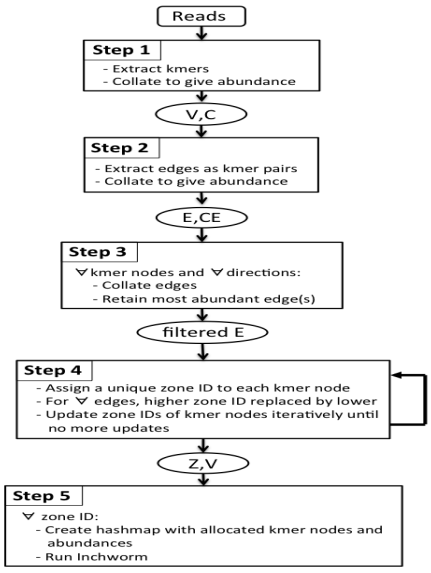

Figure 22: Steps of MapReduce-Inchworm Algorithm

In the first step, input sequence reads are decomposed into a list of unique k-mers, together with their abundances, as a single MapReduce cycle. In the second step, edges representing k-1 overlaps between k-mers are extracted in a single MapReduce operation.This pre-collection of edge information is an important feature of our algorithm. The third step filters out edges where a k-mer node has multiple candidates in the 3' or 5' directions, and is introduced to make the later Inchworm runs more robust.Inchworm builds contigs by extending a seed k-mer using the overlapping k-mer with the highest abundance, and extension continues until no more overlapping k-mers exist in the dataset. Our filtering step makes sure that the edge or edges with the highest abundance are kept in the cluster, and so available to Inchworm, while others are removed. Without this filtering operation, the subsequent step tends to produce k-mer clusters with highly diverse sizes, and leads to load balancing issues for high performance computing clusters. Having prepared the k-mer and k-mer overlap data, the fourth step performs the k-mer clustering by finding connected components, as described above.

The original C++ code of Inchworm for constructing contigs is implemented as step 5 of the algorithm, and is executed as a callback function by each set of clustered k-mers or each connected component.The input consists of two MapReduce objects, the zone assignment of k-mers from the previous step and the list of k-mers with their abundance values. These two input objects are concatenated into a single MapReduce object, followed by a collate() operation using k-mers as key. This creates KMV pairs with the k-mer as key and the pair of zone ID and abundance value as the multivalue. The following reduce() operation creates new KV pairs, this time with the zone ID as key and the corresponding pair of k-mer and abundance as the value. Another collate() operation with zone ID as key produces KMV pairs with each zone ID linked to a list of k-mer/abundant value pairs.

The final reduce() operation creates a hash_map table for each zone ID, i.e. for each cluster. This table has the k-mers V_i as keys and the abundance C_i as values. This hash_map table is an input to the Inchworm module, which constructs contigs for that cluster. The final collate() operation evenly distributes the k-mer clusters across the allocated nodes of the computer. Each compute node will then run multiple Inchworm jobs, according to the

Table 4: Number of reads and k-mers for three datasets with the computing platforms

Organism	# of reads	# of unique k-mers	MR-Inchworm Inchworm	Original	data source
mouse	105,290,476	746,811,557	iDataPlex	(64 GB)	[18]
sugarbeet	129,832,549	2,213,519,875	iDataPlex	(256 GB)	unpublished
wheat	1,468,701,119	5,775,799,648	iDataPlex	(4 TB)	unpublished

number of k-mer clusters residing on that compute node. The resulting files of Inchworm contigs can be merged for input to Chrysalis.

4.3 Results

This section presents our evaluation of MapReduce-Inchworm, in comparison to the original Inchworm. The primary aim of our work is to circumvent the high-memory requirements of the original Inchworm, while a secondary aim is to reduce the runtime required. It is vital, of course, that performance improvements do not lead to loss of accuracy, and so we begin by presenting a detailed characterization of the transcripts generated by the Trinity pipeline when MapReduce-Inchworm is used to generate the initial contigs. Next, we present performance results in terms of runtime and scalability, followed by results for the physical memory usage of MapReduce-Inchworm. Finally, we present a performance comparison using RNA-Seq datasets from several different organisms.

The datasets and computing resources used in our evaluations are listed in Table 4.

The results presented here for MapReduce-Inchworm were obtained on an IBM iDataplex-Nextscale cluster, consisting of nodes with 2 x 12-core Intel Xeon processors and 64GB of RAM. For the original version of Inchworm, the code is necessarily run on a single node, and only a single thread was used. For the mouse dataset, a single node of the iDataplex-Nextscale cluster was used. For the larger sugarbeet dataset, jobs were run on a high-memory (256GB) node of a slightly older iDataplex cluster. For the most complex transcriptome, the wheat dataset, ScaleMP software (http://www.scalemp.com/) was used to create a virtual symmetric multiprocessing node on the iDataplex cluster to meet the high memory requirement of the original Inchworm.

4.3.1 Runtime Improvement

We stratified the runtime in terms of the major steps in both versions of Inchworm, as shown in Figure 23. The original Inchworm consists of 3 principal steps: 1) jellyfish, 2) parsing k-mers, and 3) inchworm contig construction. The first step involves counting the occurrence of every unique k-mer in the set of input reads using the program Jellyfish [41], and writing the output to a disk file. In the second step, Inchworm reads the output file back into physical memory by storing each k-mer and its count into a hashmap table as a key-value pair. In the final step, the algorithm creates draft contigs using the hashmap table of unique k-mers. We divide the MapReduce-Inchworm algorithm into an initial splitting input reads step, followed by the five steps described in Methods. The initial step consists of evenly splitting the input file of reads into multiple files, according to the number of allocated compute nodes. Each file is then read into a compute node in preparation for subsequent steps.

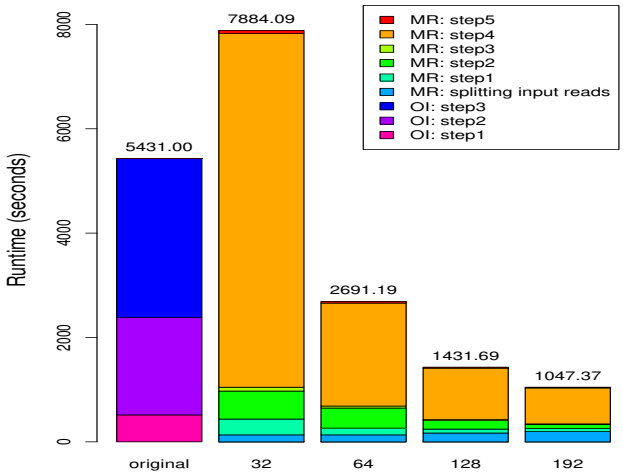

Figure 23: Comparison of runtime of the original Inchworm and the MapReduce-Inchworm

Figure 23 shows that the first two steps of the original Inchworm, which could be categorised as k-mer preparation steps, take a significant fraction of the total runtime. These steps are equivalent to the splitting input reads and step 1 of MapReduce-Inchworm. The latter steps are however much quicker because they avoid storing k-mers on disc.

The remaining runtime of the original Inchworm involves construction of contigs. In the MapReduce-Inchworm implementation, this is done individually for each cluster, and is very fast (MR: step 5 in Figure 23). The bulk of the runtime for MapReduce-Inchworm is taken by the clustering algorithm (MR: step 4 in Figure 23), and this scales well with the number of nodes used. As mentioned above, super-linear scaling is achieved in going from 32 nodes to 64 nodes because of the reduction in out-of-core processing, while going from 64 to 128 nodes gives a speedup of 1.9, and from 64 to 192 nodes a speedup of 2.6.

4.3.2 Performance comparison for RNA-Seq datasets from complex organisms

We next tested our approach on some more challenging RNA-Seq datasets obtained from sugarbeet and wheat samples (see Table 4). The memory requirement of the original Inchworm depends on the transcriptome complexity and is expected to roughly correlate with the number of unique k-mers from the input reads. The mouse, sugarbeet and wheat datasets require 46.7GB, 141.5GB and 373.9GB of memory respectively, and these values do indeed correlate with the total number of unique k-mers listed in Table 4. In fact, the required memory for the wheat dataset exceeded the physical memory on any single node of our available compute platforms. In order to run the original Inchworm, we used ScaleMP software (http://www.scalemp.com/) to aggregate 32 nodes, each providing 128GB memory, to create a vSMP node with a 4TB address space.

The runtime for the original Inchworm for sugarbeet and wheat datasets were 388.2 and 8856 minutes respectively. As mentioned before, the wheat dataset was run using ScaleMP on 32 nodes and not a single node. Using 64 nodes with MapReduce-Inchworm, the runtime of the MapReduce Inchworm was 152.7 and 1111.8 minutes respectively. With 192 nodes, using MapReduce-Inchworm, we were able to reduce the runtime to 79.2 and 381.7 minutes respectively.

In conclusion, the MapReduce-Inchworm procedure scales well to large and complex datasets. Increasing the number of compute nodes leads to a reduction in runtime, and reduced paging to disk as the per-node memory requirements are lowered. In particular, MapReduce-Inchworm allowed us to process the large wheat dataset in less t2han a day,

while the original Inchworm required an advanced platform solution to run at all.

4.4 Discussion

In this study, we enabled the parallelization of the Inchworm module of Trinity by using a MapReduce-based approach to pre-cluster the k-mers obtained from the input reads. An instance of Inchworm is run on each k-mer cluster, yielding a set of contigs per cluster. Contigs from all clusters are pooled together and passed to the Chrysalis module for re-clustering according to the original Trinity scheme. The Inchworm module of Trinity is known to be the most memory-intensive step [63], and is often a barrier to processing large or complex RNA-Seq datasets. In our scheme, the computational load is passed to the pre-clustering step, where the well-established MapReduce procedure allows the load to be distributed over a commodity compute cluster. Our approach is distinct from other recent developments, which seek to MPI-parallelise Inchworm itself (Brian Haas, personal communication).

Pell et al. [51] have introduced a Bloom filter which provides memory efficient storage of kmer graphs. Chikhi and Rizk [9] added an additional data structure holding the false positives which might arise through trial kmer extensions, as well as a marking structure holding complex nodes for use in graph traversal. In contrast, we store an explicit list of kmer nodes and edges in a set of MapReduce objects. There is no reduction in the total memory required, but rather we focus on the ability to distribute these MapReduce objects over a large number of nodes to reduce the per node memory requirement. Note that by storing edges explicitly, we do not make trial extensions from kmer nodes, which can lead to false edges in the Bloom filter method.

We expect that there should be a correspondence between the k-mer clusters we generate, and the contig clusters (components) produced by Chrysalis, in that they both relate to a set of gene products. It may be that further efficiency gains can be achieved by merging these steps, but we have not investigated this possibility in the current work, adopting instead a conservative approach. If, for example, reads from a transcribed gene yield two k-mer clusters, and hence two sets of Inchworm contigs, then the Chrysalis module should

in principle find welds between them, and recover the correct graph.

The assessment of the assembly accuracy using simulated and experimental RNA-Seq datasets shows that our parallelized Inchworm provides the final transcripts from the Trinity pipeline with marginally more accuracy compared to the original inchworm. The difference in accuracy comes from the utilization of additional edge information in MapReduce-Inchworm, which clusters k-mers guided by edge objects which link pairs of k-mers appearing consecutively in input reads. On the other hand, the original Inchworm constructs contigs directly from the set of all k-mers. Contigs are extended by searching for appropriately overlapping k-mers, rather than using pre-calculated edges. This pre-collection of edge information is feasible in our approach because of the distributed nature of the algorithm.

We have presented performance results for a range of experimental read datasets. The total runtime required to produce the complete set of Inchworm contigs could be reduced below that required for the original Inchworm, provided a moderate number of compute nodes are available. It may be debatable whether this is necessary for small datasets, such as the mouse dataset included here, but there are clearly significant gains for larger and more complex datasets (Fig. 7). More importantly, the memory requirement on each node can be reduced by distributing the job over sufficient nodes. In this way, commodity compute clusters can be used, and there is no need for high memory nodes or specialised solutions for aggregating node memory into a single address space.

4.5 Conclusion

The results of this study indicate that the MapReduce framework has great potential for processing high throughput sequencing datasets more efficiently. The proposed approach could be applied as a pre-processing step for other de novo transcriptome assemblers, by implementing the chosen assembly code as a callback function in the final reduce() step, as we have done for Inchworm in the current study. Specifically, we plan to investigate the parallelization of Oases [68] and SOAPdenovo-Trans [79] via this approach. It is also worth exploring the feasibility of the pre-clustering approach for de novo metagenome and metatranscriptome assembly, which is more complex due to the presence of multiple genomes

or transcriptomes from different species. For example, the de novo metagenome assembler [52] starts by partitioning the de Bruijn graph into isolated components corresponding to different species. Then for each component, it reconstructs the slight variants of the genomes of subspecies within the same species using multiple sequence alignments. A similar approach has been developed for de novo metatransciptome assembly [33]. Our proposed approach could be adapted to these pipelines to provide a memory-efficient method for the initial partitioning.

In conclusion, we have presented a computationally efficient method for clustering k-mers derived from RNA-Seq datasets. Applied to the Trinity pipeline, the approach avoids the large memory requirements of the original Inchworm, enabling the analysis of large datasets on commodity compute clusters. We expect that this general approach will have applications for other assembly problems.

REFERENCES

[1] Biron, I. and *et al*, "De novo transcriptome assembly with ABySS," *Bioinformatics*, vol. 25, pp. 2872–2877, 2009.

[2] Boisvert, S., Raymond, F., Godzaridis, E., Laviolette, F., and Corbeil, J., "Ray Meta: scalable de novo metagenome assembly and profiling," *Genome Biology*, vol. 13, no. 12, 2012. `http://genomebiology.biomedcentral.com/articles/10.1186/gb-2012-13-12-r122`.

[3] Boisvert, S., Laviolette, F., and Corbeil, J., "Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *Journal of Computational Biology*, vol. 17, no. 11, pp. 1519–1533, 2010.

[4] Bozdag, D., Hatem, A., and Catalyurek, U. V., "Exploring parallelism in short sequence mapping using Burrows-Wheeler Transform," in *Proc. Int'l Parallel and Distributed Processing Symp. Workshops and Phd forum (IPDPSW 2010)*, (Atlanta, GA), pp. 1–8, Apr. 2010.

[5] Brown, C., Howe, A., Zhang, Q., Pyrkosz, A., and Brom, T., "A reference-free algorithm for computational normalization of shotgun sequencing data," *arXiv:12034802*, 2012.

[6] Cabau, C. and *et al*, "Compacting and correcting Trinity and Oases RNA-Seq de novo assemblies," *PeerJ*, vol. 5, 2017.

[7] Carrier, P., Long, B., Walsh, R., Dawson, J., Haas, B., Trickle, T., Sosa, C. P., and William, T., "The impact of high-performance computing best practice applied to next-generation sequencing workflows." `http://biorxiv.org/content/biorxiv/early/2015/04/07/017665.full.pdf`.

[8] Chang, Z. and *et al*, "Bridger: a new framework for de novo transcriptome assembly using RNA-seq data," *Genome Research*, vol. 20, pp. 265–272, 2010.

[9] Chikhi, R. and Rizk, G., "Space-efficient and exact de Bruijn graph representation based on a Bloom filter," *Algorithms for Molecular Biology*, vol. 8, 2013.

[10] Collectl, "Collectl Tutorial - The Basics." `http://collectl.sourceforge.net`.

[11] Corney, D., "RNA-Seq using next generation sequencing," *Materials and Methods*, vol. 3, pp. 644–652, 2013.

[12] Dean, J. and Ghemawat, S., "Simplified data processing on large clusters," *Computing in Science and Engineering*, pp. 29–41, 2012.

[13] Dubinkina, V. B., Ischenko, D. S., Ulyantsev, V., Tyakht, A. V., and Alexeev, D. G., "Assessment of k-mer spectrum applicability for metagenomic dissimilarity analysis," *BMC Bioinformatics*, vol. 17, no. 38, 2016.

[14] Evolution and Genomics, "Whitefly dataset." http://evomics.org/learning/genomics/trinity/.

[15] Fullwood, M., Wei, C.-L., Liu, E., and Ruan, Y., "Next-generation DNA sequencing of paired-end tags (PET) for transcriptome and genome analyses," *Genome Research*, vol. 4, pp. 521–532, 2009.

[16] Genomics England, "100k genome project." WWW page. http://www.genomicsengland.co.uk/100k-genome-project/.

[17] Georganas, E., Buluç, A., Chapman, J., Oliker, L., Rokhsar, D., and Yelick, K., "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2014)*, (New Orleans, LA), Nov. 2014.

[18] Grabherr, M. G. and *et al*, "Full-length transcriptome assembly from RNA-Seq data without a reference genome," *Nature BioTechnology*, vol. 29, pp. 644–652, 2011.

[19] Grambron, P., Winn, M., Venables, N., Sachdeva, V., Rowe, W., and Jordan, K., "Assembly-free metagenomic analyses on distributed computational platforms," *Proc. of the Rocky Mountain Bioinformatics 2017*, 2017.

[20] Gunaratne, P., Coarfa, C., Soibam, B., and Tandon, A., "miRNA data analysis: next-gen sequencing," *Methods Mol Biology*, vol. 822, pp. 273–288, 2012.

[21] Haas, B. J. and *et al*, "De novo transcript sequence reconstruction from RNA-seq using the trinity platform for reference generation and analysis," *Nature Protocols*, vol. 8, pp. 1494–1512, 2013.

[22] Haring, R. A., Ohmacht, M. A., Fox, T. W., Gschwind, M. K., Satterfield, D. L., Sugavanam, K., Coteus, P. W., Heidelberger, P., Blumrich, M. A., Wisniewski, R. W., Gara, A., Chiu, G. L.-T., Boyle, P. A., Chist, N. H., and Kim, C., "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.

[23] Henschel, R., Lieber, M., Wu, L.-S., Nasta, P. M., Haas, B. J., and LeDuc, R. D., "Trinity RNA-Seq assembler performance optimization," in *Proc. 1st Conference of the Extreme Science and Engineering Discovery Environment (XSEDE)*, (Chicago, IL), 2012.

[24] Hopcroft, J. and Tarjan, R., "Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, pp. 372–378, 1973.

[25] Illie, L. and Molnar, M., "RACER: Rapid and accurate correction of errors in reads," *Bioinformatics*, vol. 29, no. 19, pp. 2490–2493, 2013.

[26] Inst., B., "RNA-Seq De novo Assembly using Trinity." http://trinityrnaseq.sourceforge.net.

[27] Jackson, B. G., Schnable, P. S., and Aluru, S., "Parallel short sequence assembly of transcriptomes," *BMC Bioinformatics*, vol. 10, no. Suppl 1, 2009.

[28] Jammula, N., Chockalingam, S., and Aluru, S., "Parallel Read Error Correction for Big Genomic Datasets," in *Proc. Int'l Conf. High-Performance Computing (HiPC)*, (Bengaluru, India), IEEE Press, Dec. 2015.

[29] Kamil, S., Shalf, J., and Strohmaier, E., "Power efficiency in high performance computing," in *Proceedings of the IEEE International Distributed and Processing Symposium*, (Miami, FL), Apr. 2008.

[30] Kao, W.-C., Chan, A. H., and Song, Y. S., "ECHO: A reference-free short-read error correction algorithm," *Genome Research*, vol. 21, pp. 1181–1192, 2011.

[31] Kelley, D. R., Schatz, M. C., and Salzberg, S. L., "Quake: quality-aware detection and correction of sequencing errors," *Genome Biology*, vol. 11, no. 11, 2010. `http://www.genomebiology.com/2010/11/11/R116`.

[32] Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L., "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. R25, 2009.

[33] Leung, H., Yiu, S., Parkinson, J., and Chin, F., "IDBA-MT: De Novo Assembler for Metatranscriptomic Data Generated from Next-Generation Sequencing Technology," *J Comp Biol*, vol. 20, pp. 540–550, 2013.

[34] Li, B. and Dewey, C., "RSEM: accurate transcript quantification from RNA-Seq data with or without a reference genome," *BMC Bioinformatics*, vol. 12, 2011.

[35] Li, H., Handsaker, B., and Wysoker, A., "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[36] Li, R. and *et al*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome Research*, vol. 20, pp. 265–272, 2010.

[37] Liu, J. and *et al*, "BinPacker: Packing-Based De Novo Transcriptome Assembly from RNA-seq Data," *PLOS Computational Biology*, vol. 12, no. e1004772, 2016.

[38] Liu, Y., Schmidt, B., and Maskell, D. L., "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI," *BMC Bioinformatics*, vol. 12, 2011.

[39] Macmanes, M. D. and Eisen, M. B., "Improving transcriptome assembly through error correction of high-throughput sequence reads," *PeerJ*, p. e113, 2013. `https://peerj.com/articles/113/`.

[40] Magoulas, R., "Big data." `http://strata.oreilly.com/2010/01/roger-magoulas-on-big-data.html`.

[41] Marcais, G. and Kingsford, C., "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.

[42] Martello, S. and Toth, P., eds., *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[43] Martin, J. A. and Wang, Z., "Next-generation transcriptome assembly," *Nature Reviews Genetics*, vol. 12, pp. 671–682, 2011.

[44] McKenna, A. and *et al*, "The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, pp. 1297–1303, 2010.

[45] Memczak, S. and *et al*, "Circular RNAs are a large class of animal RNAs with regulatory potency," *Nature*, vol. 495, pp. 333–338, 2013.

[46] Mohammed, E., Far, B., and Naugler, C., "Applications of the MapReduce programming framework to clinical big data analysis: current landscape and future trends," *Biodata Mining*, vol. 7, no. 22, 2014.

[47] Morin, R. and *et al*, "Marra MA: Profiling the HeLa S3 transcriptome using randomly primed cDNA and massively parallel short-read sequencing," *Biotechniques*, vol. 45, pp. 81–94, 2008.

[48] Myers, E. W., "Toward simplifying and accurately formulating fragment assembly," *Journal of Computational Biology*, pp. 275–290, 1995.

[49] Oshlack, A., Robinson, M., and Young, M., "From RNA-Seq reads to differential expression results," *Genome Biology*, vol. 11, 2010.

[50] Pearson, W. R., "Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms.," *Genomics*, vol. 11, no. 3, pp. 635–650, 1991.

[51] Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J., and Browne, C., "Scaling metagenome sequence assembly with probabilistic de Bruijn graphs," *Proc. National Academy Sci.*, vol. 109, pp. 13272–13277, 2012.

[52] Peng, Y., Leung, H., Yiu, S., and Chin, F., "Meta-IDBA: a de novo assembler for metagenomic data," *Bioinformatics*, vol. 27, pp. 94–101, 2011.

[53] Pevzner, P. A., Tang, H., and Waterman, M. S., "An eulerian path approach to dna fragment assembly," in *Proceedings of the National Academy of Sciences*, vol. 1995, pp. 9748–9753, 2001.

[54] Plimpton, S. J., "Mapreduce-mpi library." http://mapreduce.sandia.gov/papers.html, 2011.

[55] Plimpton, S. and Devine, K., "MapReduce in MPI for Large-Scale Graph Algorithms," *Parallel Computing*, vol. 37, pp. 610–632, 2011.

[56] Prufer, K. and *et al*, "The complete genome sequence of a neanderthal from the altai mountains," *Nature*, 2013.

[57] "PyFasta: A fast, memory-efficient, pythonic (and command-line) access to fasta sequence files." `https://pypi.python.org/pypi/pyfasta/`.

[58] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C., "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proc. 13th Int'l Symp. on High Performance Computer Architecture*, pp. 13–24, 2007.

[59] Reddy, A., Rogers, M., Richardson, D., Hamilton, M., and Ben-Hur, A., "Deciphering the plant splicing code: experimental and computational approaches for predicting alternative splicing and splicing regulatory elements," *Frontiers in Plan Science*, vol. 3, 2012.

[60] Richter, D., Ott, F., Auch, A., Schmid, R., and Huson, D., "MetaSimA Sequencing Simulator for Genomics and Metagenomics," *PLoS ONE*, vol. 3, no. 10, 2008. https://doi.org/10.1371/journal.pone.0003373.

[61] Rizk, G., Lavenier, D., and Chikhi, R., "Dsk: k-mer counting with very low memory usage.," *BMC Bioinformatics*, vol. 29, no. 5, pp. 652–653, 2013.

[62] Robinson, M. D., McCarthy, D. J., and Smyth, G. K., "edgeR:a Bioconductor package for differential expression analysis of digital gene expression data," *BMC Genomics*, vol. 26, no. 1, pp. 139–140, 2009.

[63] Sachdeva, V., Kim, C., Jordan, K., and Winn, M., "Parallelization of the Trinity pipeline for de novo transcriptome assembly," *Parallel and Distributed Processing Symposium Workshops*, pp. 566–575, 2014.

[64] Sanger Trust Institute, "Cancer genome project." WWW page. `http://www.sanger.ac.uk/research/projects/cancergenome/`.

[65] Schrder, J., Schrder, H., Puglisi, S. J., Sinha, R., and Schmidt, B., "SHREC: a short-read error correction method," *Bioinformatics*, vol. 25, pp. 2157–2163, 2009.

[66] Schulz, M. H., Weese, D., Holtgrewe, M., Dimitrova, V., Niu, S., Reinert, K., and Richard, H., "Fiona: a parallel and automatic strategy for read error correction," *Bioinformatics*, vol. 30, pp. 356–363, 2014.

[67] Schulz, M., Zerbino, D., Vingron, M., and Birney, E., "Oases: Robust de novo RNA-seq assembly across the dynamic range of expression levels," *Bioinformatics*, vol. 28, pp. 1086–1092, 2012.

[68] Schulz, M., Zerbino, D., Vingron, M., and Birney, E., "Oases: Robust de novo RNA-seq assembly across the dynamic range of expression levels," *Bioinformatics*, vol. 28, pp. 1086–1092, 2012.

[69] Schuster, S. C., "Next-generation sequencing transforms today's biology," *Nature*, vol. 5, pp. 16–18, 2008.

[70] Shah, A. R., Chockalingam, S., and Aluru, S., "A parallel algorithm for spectrum-based short read error correction," in *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, (Shanghai, CN), pp. 60–70, May 2012.

[71] Simpson, J. and *et al*, "ABySS: A parallel assembler for short read sequence data," *Genome Research*, vol. 19, pp. 1117–1123, 2009.

[72] Simpson, J., Wong, K., Schein, J., and Birol, S. J. I., "ABySS: a parallel assembler for short read sequence data," *Genome Research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[73] Steijger, T., Abril, J. F., Engstrm, P. G., Kokocinski, F., Hubbard, T. J., Guig, R., Harrow, J., and Bertone, P., "Assessment of transcript reconstruction methods for rna-seq," *Nature Methods*, vol. 10, pp. 1177–1184, 2013.

[74] Sze, S.-H. and Tarone, A., "A memory-efficient algorithm to obtain splicing graphs and de novo expression estimates from de Bruijn graphs of RNA-Seq data," *BMC Genomics*, vol. 15, no. S6, 2014.

[75] Ulitsky, I. and Bartel, D., "lincRNAs: genomics, evolution, and mechanisms," *Cell*, vol. 154, pp. 26–46, 2013.

[76] Wang, Y. and Smith, S. A., "Optimizing de novo assembly of short-read RNA-seq data for phylogenomics," *BMC Genomics*, vol. 14, no. 328, 2013.

[77] White, T., *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[78] Wirawan, A., Harris, R. S., Liu, Y., Schmidt, B., and Schrder, J., "HECTOR: a parallel multistage homopolymer spectrum based error corrector for 454 sequencing data," *BMC Bioinformatics*, vol. 15, 2014.

[79] Xie, Y. and *et al*, "SOAPdenovo-Trans: De novo transcriptome assembly with short RNA-Seq reads," *Bioinformatics*, vol. 30, pp. 1660–1666, 2014.

[80] Xu, B., Gao, J., and Li, C., "An efficient algorithm for DNA fragment assembly in MapReduce," *Biochem Biophys Res Commun*, vol. 426, pp. 395–398, 2012.

[81] Yang, X., Chockalingam, S. P., and Aluru, S., "A survey of error-correction methods for next-generation sequencing," *Briefings in Bioinformatics*, vol. 14, no. 1, pp. 56–66, 2013.

[82] Yang, X., Dorman, K. S., and Aluru, S., "Reptile: representative tiling for short read error correction," *Bioinformatics*, vol. 26, no. 20, pp. 2526–2533, 2010.

[83] Zerbino, D. and Birney, E., "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs," *Genome Research*, vol. 18, pp. 821–829, 2008.

[84] Zhao, Q.-Y., Wang, Y., Kong, Y.-M., Luo, D., Li, X., and Hao, P., "Optimizing de novo transcriptome assembly from short-read RNA-Seq data: a comparative study," *BMC Bioinformatics*, vol. 12, 2011.

[85] Zhao, Q.-Y., Wang, Y., Kong, Y.-M., Luo, D., Li, X., and Hao, P., "Optimizing de novo transcriptome assembly from short-read RNA-Seq data: a comparative study," *BMC Bioinformatics*, vol. 12, no. Suppl 14, 2011.